КЛАССИКА COMPUTER SCIENCE

АРХИТЕКТУРА КОМПЬЮТЕРА

4-Е ИЗДАНИЕ



Э. ТАНЕНБАУМ

PH PTR

ПИТЕР®

СЕРИЯ





STRUCTURED COMPUTER ORGANIZATION

Fourth Edition

Andrew S.Tanenbaum



Prentice Hall PTR
Upper Saddle River, New Jersey 07458
www.phptr.com

КЛАССИКА COMPUTER SCIENCE

Э. ТАНЕНБАУМ

АРХИТЕКТУРА КОМПЬЮТЕРА

4-Е ИЗДАНИЕ

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск
2005

ББК 32.973.2-02
УДК 681.3
Т18

Таненбаум Э.
Т18 Архитектура компьютера. 4-е изд. — СПб.: Питер, 2005. — 699 с.: ил. — (Серия «Классика computer science»).

ISBN 5-318-00298-6

Книга, вышедшая уже в четвертом издании, посвящена структурной организации компьютера. В ее основе лежит идея о том, что компьютер можно рассматривать как иерархию уровней, каждый из которых выполняет какую-либо определенную функцию. В рамках этого нетрадиционного подхода подробно описываются цифровой логический уровень, уровень архитектуры команд, уровень операционной системы и уровень языка ассемблера.

Книга расчитана на широкий спектр читателей — от студентов, изучающих компьютерные технологии, до тех, кто самостоятельно знакомится с архитектурой компьютера.

ББК 32.973.2-02
УДК 681.3

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 0-13-095990-1 (англ.)
ISBN 5-318-00298-6

© 1999,1990,1984,1976 by Prentice Hall, Inc.
© Перевод на русский язык, ЗАО Издательский дом «Питер», 2003
© Издание на русском языке, оформление, ЗАО Издательский дом «Питер», 2005

Краткое содержание

Предисловие 15

Глава 1. Предисловие 18

Глава 2. Организация компьютерных систем 56

Глава 3. Цифровой логический уровень 139

Глава 4. Микроархитектурный уровень 230

Глава 5. Уровень архитектуры команд 334

Глава 6. Уровень операционной системы 437

Глава 7. Уровень языка ассемблера 517

Глава 8. Архитектуры компьютеров параллельного действия 556

Глава 9. Библиография 647

Приложение А. Двоичные числа 663

Приложение Б. Числа с плавающей точкой 674

Алфавитный указатель 683

Содержание

Об авторе 14
Предисловие 15

Глава 1. Предисловие 18

Многоуровневая компьютерная организация 18
Языки, уровни и виртуальные машины 19
Современные многоуровневые машины 21
Развитие многоуровневых машин 24
Развитие компьютерной архитектуры 29
Нулевое поколение — механические компьютеры (1642–1945) 29
Первое поколение — электронные лампы (1945–1955) 32
Второе поколение — транзисторы (1955–1965) 35
Третье поколение — интегральные схемы (1965–1980) 37
Четвертое поколение — сверхбольшие интегральные схемы (1980–?) 39
Типы компьютеров 40
Технологические и экономические аспекты 41
Широкий спектр компьютеров 42
Семейства компьютеров 45
Pentium II 45
UltraSPARC II 48
PicoJava II 50
Краткое содержание книги 52
Вопросы и задания 54

Глава 2. Организация компьютерных систем 56

Процессоры 56
Устройство центрального процессора 57
Выполнение команд 58
RISC и CISC 62
Принципы разработки современных компьютеров 64
Параллелизм на уровне команд 65
Параллелизм на уровне процессоров 69
Основная память 73
Бит 73
Адреса памяти 74
Упорядочение байтов 75
Код с исправлением ошибок 77
Кэш-память 81
Сборка модулей памяти и их типы 84

Вспомогательная память ... 85
Иерархическая структура памяти ... 85
Магнитные диски ... 87
Дискеты ... 90
Диски IDE ... 91
SCSI-диски ... 92
RAID-массивы ... 93
Компакт-диски ... 98
CD-R ... 102
CD-RW ... 105
DVD ... 105
Процесс ввода-вывода ... 108
Шины ... 108
Терминалы ... 111
Мыши ... 119
Принтеры ... 121
Модемы ... 126
Коды символов ... 129
Краткое содержание главы ... 133
Вопросы и задания ... 134

Глава 3. Цифровой логический уровень ... 139

Вентили и булева алгебра ... 139
Вентили ... 139
Булева алгебра ... 142
Реализация булевых функций ... 144
Эквивалентность схем ... 145
Основные цифровые логические схемы ... 149
Интегральные схемы ... 149
Комбинационные схемы ... 151
Арифметические схемы ... 157
Тактовые генераторы ... 161
Память ... 163
Защелки ... 163
Триггеры (flip-flops) ... 165
Регистры ... 168
Организация памяти ... 168
Микросхемы памяти ... 172
ОЗУ и ПЗУ ... 174
Микросхемы процессоров и шины ... 177
Микросхемы процессоров ... 177
Шины ... 179
Ширина шины ... 182
Синхронизация шины ... 183
Арбитраж шины ... 188
Принципы работы шины ... 191
Примеры центральных процессоров ... 193
Pentium II ... 193
UltraSPARC II ... 200
PicoJava II ... 203

Примеры шин ... 205
Шина ISA ... 206
Шина PCI ... 207
Шина USB ... 215
Средства сопряжения ... 219
Микросхемы ввода-вывода ... 219
Декодирование адреса ... 220
Краткое содержание главы ... 223
Вопросы и задания ... 224

Глава 4. Микроархитектурный уровень ... 230

Пример микроархитектуры ... 230
Тракт данных ... 231
Микрокоманды ... 237
Управление микрокомандами: Mic-1 ... 240
Пример архитектуры команд: IJVM ... 244
Стек ... 245
Модель памяти IJVM ... 247
Набор команд IJVM ... 248
Компиляция Java для IJVM ... 252
Пример реализации микроархитектуры ... 254
Микрокоманды и их запись ... 254
Реализация IJVM с использованием Mic-1 ... 258
Разработка микроархитектурного уровня ... 271
Скорость и стоимость ... 271
Сокращение длины пути ... 274
Микроархитектура с упреждающей выборкой команд из памяти: Mic-2 ... 280
Конвейерная архитектура: Mic-3 ... 284
Конвейер с 7 стадиями: Mic-4 ... 290
Увеличение производительности ... 293
Кэш-память ... 294
Прогнозирование ветвления ... 300
Исполнение с изменением последовательности и подмена регистров ... 306
Спекулятивное выполнение ... 311
Примеры микроархитектурного уровня ... 314
Микроархитектура процессора Pentium II ... 314
Микроархитектура процессора UltraSPARC II ... 319
Микроархитектура процессора picoJava II ... 322
Сравнение Pentium, UltraSPARC и picoJava ... 327
Краткое содержание главы ... 329
Вопросы и задания ... 330

Глава 5. Уровень архитектуры команд ... 334

Общий обзор уровня архитектуры команд ... 336
Свойства уровня команд ... 336
Модели памяти ... 338
Регистры ... 340
Команды ... 342
Общий обзор уровня команд машины Pentium II ... 342
Общий обзор уровня команд системы UltraSPARC II ... 345
Общий обзор виртуальной машины Java ... 348

Типы данных ... 349
Числовые типы данных ... 350
Нечисловые типы данных ... 351
Типы данных процессора Pentium II ... 351
Типы данных машины UltraSPARC II ... 352
Типы данных виртуальной машины Java ... 352
Форматы команд ... 353
Критерии разработки для форматов команд ... 354
Расширение кода операций ... 356
Форматы команд процессора Pentium II ... 358
Форматы команд процессора UltraSPARC II ... 360
Форматы команд JVM ... 361
Адресация ... 364
Способы адресации ... 365
Непосредственная адресация ... 365
Прямая адресация ... 366
Регистровая адресация ... 366
Косвенная регистровая адресация ... 366
Индексная адресация ... 367
Относительная индексная адресация ... 369
Стековая адресация ... 369
Способы адресации для команд перехода ... 372
Ортогональность кодов операций и способов адресации ... 373
Способы адресации процессора Pentium II ... 375
Способы адресации процессора UltraSPARC II ... 377
Способы адресации машины JVM ... 377
Сравнение способов адресации ... 378
Типы команд ... 379
Команды перемещения данных ... 379
Бинарные операции ... 380
Унарные операции ... 381
Сравнения и условные переходы ... 383
Команды вызова процедур ... 385
Управление циклом ... 385
Команды ввода-вывода ... 386
Команды процессора Pentium II ... 390
Команды UltraSPARC II ... 394
Команды компьютера picoJava II ... 397
Сравнение наборов команд ... 403
Поток управления ... 404
Последовательный поток управления и переходы ... 404
Процедуры ... 405
Сопрограммы ... 410
Ловушки ... 412
Прерывания ... 413
Ханойская башня ... 417
Решение задачи «Ханойская башня» на ассемблере Pentium II ... 418
Решение задачи «Ханойская башня» на ассемблере UltraSPARC II ... 419
Решение задачи «Ханойская башня» на ассемблере для JVM ... 421

Intel IA-64 423
Проблема с Pentium II 423
Модель IA-64: открытое параллельное выполнение команд 425
Предикация 426
Спекулятивная загрузка 429
Проверка в реальных условиях 430
Краткое содержание главы 430
Вопросы и задания 431

Глава 6. Уровень операционной системы 437

Виртуальная память 438
Страничная организация памяти 439
Реализация страничной организации памяти 441
Вызов страниц по требованию и рабочее множество 444
Политика замещения страниц 445
Размер страниц и фрагментация 448
Сегментация 449
Как реализуется сегментация 452
Виртуальная память в процессоре Pentium II 455
Виртуальная память UltraSPARC II 460
Виртуальная память и кэширование 463
Виртуальные команды ввода-вывода 463
Файлы 464
Реализация виртуальных команд ввода-вывода 465
Команды управления директориями 469
Виртуальные команды для параллельной обработки 470
Формирование процесса 471
Состояние гонок 472
Синхронизация процесса с использованием семафоров 476
Примеры операционных систем 479
Введение 480
Примеры виртуальной памяти 489
Примеры виртуального ввода-вывода 493
Примеры управления процессами 504
Краткое содержание главы 510
Вопросы и задания 511

Глава 7. Уровень языка ассемблера 517

Введение в язык ассемблера 518
Что такое язык ассемблера? 518
Зачем нужен язык ассемблера? 519
Формат оператора в языке ассемблера 521
Директивы 524
Макросы 527
Макроопределение, макровызов и макрорасширение 527
Макросы с параметрами 529
Расширенные возможности 530
Реализация макросредств в ассемблере 530
Процесс ассемблирования 531
Двухпроходной ассемблер 531
Первый проход 532

Второй проход ... 536
Таблица символов ... 537
Связывание и загрузка ... 538
Задачи компоновщика ... 540
Структура объектного модуля ... 543
Время принятия решения и динамическое перераспределение памяти ... 545
Динамическое связывание ... 547
Краткое содержание главы ... 551
Вопросы и задания ... 552

Глава 8. Архитектуры компьютеров параллельного действия ... 556

Вопросы разработки компьютеров параллельного действия ... 557
Информационные модели ... 559
Сети межсоединений ... 564
Производительность ... 572
Метрика программного обеспечения ... 574
Программное обеспечение ... 579
Классификация компьютеров параллельного действия ... 584
Компьютеры SIMD ... 587
Массивно-параллельные процессоры ... 587
Векторные процессоры ... 588
Мультипроцессоры с памятью совместного использования ... 592
Семантика памяти ... 593
Архитектуры UMA SMP с шинной организацией ... 597
Мультипроцессоры UMA с координатными коммутаторами ... 603
Мультипроцессоры UMA с многоступенчатыми сетями ... 605
Мультипроцессоры NUMA ... 607
Мультипроцессоры CC-NUMA ... 609
Мультипроцессоры COMA ... 619
Мультикомпьютеры с передачей сообщений ... 621
MPP — процессоры с массовым параллелизмом ... 622
COW — Clusters of Workstations (кластеры рабочих станций) ... 626
Планирование ... 627
Связное программное обеспечение для мультикомпьютеров ... 632
Совместно используемая память на прикладном уровне ... 635
Краткое содержание главы ... 642
Вопросы и задания ... 643

Глава 9. Библиография ... 647

Литература для дальнейшего чтения ... 647
Организация компьютерных систем ... 648
Цифровой логический уровень ... 649
Микроархитектурный уровень ... 649
Уровень команд ... 650
Уровень операционной системы ... 651
Уровень языка ассемблера ... 652
Архитектуры компьютеров параллельного действия ... 652
Двоичные числа и числа с плавающей точкой ... 653
Алфавитный список литературы ... 654

Приложение А. Двоичные числа 665

Числа конечной точности 665
Позиционные системы счисления 667
Преобразование чисел из одной системы счисления в другую 669
Отрицательные двоичные числа 670
Двоичная арифметика 673
Вопросы и задания 674

Приложение Б. Числа с плавающей точкой 676

Принципы представления с плавающей точкой 676
Стандарт IEEE 754 680
Вопросы и задания 683

Алфавитный указатель 685

Сюзанне, Барбаре, Марвину, Брэму и памяти моей дорогой π

Об авторе

Эндрю С. Таненбаум получил степень бакалавра естественных наук в Массачусетском технологическом институте и степень доктора в Университете Калифорнии в Беркли. В настоящее время является профессором Амстердамского университета в Нидерландах, где возглавляет группу разработчиков компьютерных систем. Он также возглавляет факультет вычислительной техники (межвузовскую аспирантуру, в которой исследуются и разрабатываются системы параллельной обработки, распределенные системы и системы формирования изображения). Тем не менее он всеми силами старается не превратиться в бюрократа.

В прошлом он занимался компиляторами, операционными системами, сетями и локальными распределенными системами. Его настоящее исследование связано с разработкой глобальных распределенных систем, которые включают в себя миллионы пользователей. Результатом этих исследовательских проектов стали 85 статей, опубликованных в разных журналах, выступления на конференциях и 5 книг.

Профессор Таненбаум разрабатывает программное обеспечение. Он является главным разработчиком пакета «Amsterdam Compiler Kit» (набора инструментальных средств для написания портативных компиляторов), а также разработчиком системы MINIX (клона UNIX для студенческих лабораторий программирования). Вместе со своими учениками и программистами он участвовал в разработке системы Amoeba (это распределенная система с высокой производительностью на основе микроядра). Системы MINIX и Amoeba находятся в свободном доступе в Интернете.

Его аспиранты достигли больших высот после получения ученых степеней. Он очень гордится ими.

Профессор Таненбаум — член Ассоциации по вычислительной технике, член Института инженеров по электротехнике и электронике (IEEE), член Королевской голландской академии науки и искусства. В 1994 году получил премию от Ассоциации по вычислительной технике как выдающийся педагог. В 1997 году награжден премией от Специальной группы по образованию в области вычислительной техники (Ассоциации по вычислительной технике) за вклад в образование в области вычислительной техники. Его имя включено в справочник «*Кто есть кто*» («*Who's Who in the World*»). Его домашнюю страничку в Интернете можно найти по адресу http://www.cs.vu.nl/~ast/.

Предисловие

В основе первых трех изданий книги лежит идея о том, что компьютер можно рассматривать как иерархию уровней, каждый из которых выполняет какую-либо определенную функцию. Это фундаментальное утверждение сейчас столь же правомерно, как и в момент выхода в свет первого издания, поэтому мы по-прежнему берем его за основу, на этот раз уже в четвертом издании. Как и в первых трех изданиях, в этой книге мы подробно описываем цифровой логический уровень, уровень архитектуры команд, уровень операционной системы и уровень языка ассемблера (хотя мы изменили некоторые названия, чтобы следовать современным установившимся обычаям).

В целом структура книги осталась прежней, но в четвертое издание внесены некоторые изменения, что объясняется стремительным развитием компьютерной промышленности. Например, все программы, которые в предыдущих изданиях были написаны на языке Pascal, в четвертом издании переписаны на язык Java, чтобы продемонстрировать популярность языка Java в настоящее время. Кроме того, в качестве примеров в книге рассматриваются более современные машины (Intel Pentium II, Sun UltraSPARC II и Sun picoJava II).

Мультипроцессоры и компьютеры параллельного действия получили широкое распространение, поэтому материал, связанный с архитектурами параллельного действия, был полностью переделан и значительно расширен. В этой книге мы затрагиваем широкий диапазон тем от мультипроцессоров до кластеров рабочих станций.

С годами книга увеличилась в объеме (хотя не так сильно, как другие популярные компьютерные издания). Это неизбежно, поскольку происходит постоянное развитие и о предмете становится известно все больше и больше. Поэтому если книга используется в целях обучения, нужно иметь в виду, что этот курс может занять более длительное время, чем раньше. Возможный вариант — изучать первые три главы, часть четвертой главы (до раздела о разработке микроархитектурного уровня включительно) и пятую главу в качестве минимума, а оставшееся время на ваше усмотрение потратить на шестую, седьмую, восьмую главы и вторую часть четвертой главы.

В четвертое издание внесены следующие изменения. В главе 1 по-прежнему излагается история развития архитектуры компьютеров, но мы расширили ряд рассматриваемых машин. В главе вводятся три основных примера: Pentium II, UltraSPARC II и picoJava II.

Материал второй главы обновлен и переработан. В ней мы рассматриваем современные устройства ввода-вывода: диски RAID, CD-R, DVD, цветные принтеры и т. п.

Глава 3 (цифровой логический уровень) претерпела некоторые изменения — теперь в ней рассматриваются компьютерные шины и современные устройства ввода-вывода. Главное изменение — это новый материал о шинах (в частности, PCI и USB). Три новых примера описываются на уровне микросхем.

Глава 4 (теперь она называется «Микроархитектурный уровень») была полностью переписана. Идея использовать пример микропрограммируемой машины для демонстрации работы тракта данных была сохранена, но в качестве примера взят сокращенный вариант JVM. В соответствии с этим была изменена микроархитектура. В главе продемонстрированы возможные компромиссы с точки зрения стоимости и производительности. В последнем примере, Mic-4, используется конвейер из семи стадий. Этот пример наглядно демонстрирует основные принципы работы современных компьютеров (например, Pentium II). К главе добавлен новый раздел о способах увеличения производительности, в котором рассматриваются новые технологии (кэширование, прогнозирование переходов, исполнение с изменением последовательности, спекулятивное выполнение и предикация). Новые примеры машин рассматриваются на микроархитектурном уровне.

В главе 5 (теперь она называется «Уровень архитектуры команд») рассказывается о так называемом машинном языке. В качестве основных примеров здесь используются Pentium II, UltraSPARC II и JVM.

Глава 6 (уровень операционной системы) содержит примеры операционных систем для Pentium II (Windows NT) и UltraSPARC II (UNIX). Первая операционная система сравнительно новая. Она содержит множество особенностей, которые стоит изучить. Система UNIX все еще используется во многих университетах и компаниях, и, кроме того, она довольно проста, поэтому тоже заслуживает нашего внимания.

В главе 7 (уровень языка ассемблера) появились примеры для тех машин, которые мы рассматриваем в этой книге. Кроме того, добавлен новый материал о динамическом связывании.

Глава 8 (архитектура компьютеров параллельного действия) полностью изменена. В ней подробно описываются мультипроцессоры (UMA, NUMA и COMA) и мультикомпьютеры (MPP и COW).

Пересмотрен список литературы. Большинство работ, на которые мы ссылаемся в этой книге, опубликованы после выхода третьего издания. Двоичные числа и числа с плавающей точкой не сильно изменились за последнее время, поэтому приложения вошли в четвертое издание почти без изменений.

Наконец, многие проблемы, изложенные в третьем издании, были пересмотрены и к ним добавлены новые.

Существует web-сайт этой книги. В Интернете имеются файлы PostScript для всех иллюстраций. Их можно получить и распечатать. Кроме того, там можно найти симулятор и другие инструментальные программные средства. Универсальный адрес ресурса: http://www.cs.vu.nl/~ast/sco4/

Симулятор и программные средства написаны Реем Онтко (Ray Ontko). Автор выражает признательность Рею за эти чрезвычайно полезные программы.

Автор искренне благодарит всех, кто читал рукопись данной книги и высказал ценные замечания и предложения или оказал какую-либо помощь, в частности Генри Бола (Henri Bal), Алана Чарльзуорта (Alan Charlesworth), Куроша Гарахор-

лоо (Koorosh Gharachorloo), Маркуса Гонкалвеса (Marcus Goncalves), Карена Панетту Ленц (Karen Panetta Lentz), Тимоти Мэттсона (Timothy Mattson), Гарлана Мак-Гана (Harlan McGhan), Майлза Мердокка (Miles Murdocca), Кэвина Нормойла (Kevin Normoyle), Майка О'Коннора (Mike O'Connor), Митсунори Огихара (Mitsunori Ogihara), Рея Онтко (Ray Ontko), Аске Плаата (Aske Plaat), Вильяма Потвина (William Potvin II), Нагарайана Прабхакарана (Nagarajan Prabhakaran), Джеймса Г. Пагсли (James H. Pugsley), Рональда Н. Шредера (Ronald N. Schroeder), Райана Шумейкера (Ryan Shoemaker), Чарльза Силио-мл. (Charles Silio, Jr.) и Дейла Скрина (Dale Skrien). Мои ученики Адриаан Бон (Adriaan Bon), Лаура де Вриес (Laura de Vries), Дольф Лот (Dolf Loth), Гуидо ван Нордент (Guido van't Noordende) помогали мне в работе над текстом, за что им большое спасибо.

Особую благодарность выражаю Джиму Гудману (Jim Goodman) за его вклад в создание этой книги (в частности, четвертой и пятой глав). Идея использовать JVM принадлежит именно ему, и микроархитектура для реализации JVM тоже его. Он же предложил множество новаторских идей. Книга значительно улучшилась благодаря его содействию.

Наконец, я хотел бы поблагодарить Сюзанну за терпеливое отношение ко мне, несмотря на то, что я долгие часы проводил за своим Pentium'ом. С моей точки зрения, Pentium — более усовершенствованная машина, чем мой старый IBM-386, но с ее точки зрения никакой разницы нет. Я также хочу поблагодарить Барбару и Марвина за то, что они такие замечательные дети, а также Брэма за то, что он вел себя тихо, когда я писал эту книгу.

Эндрю С. Таненбаум

Глава 1
Предисловие

Цифровой компьютер — это машина, которая может решать задачи, выполняя данные ей команды. Последовательность команд, описывающих решение определенной задачи, называется **программой**. Электронные схемы каждого компьютера могут распознавать и выполнять ограниченный набор простых команд. Все программы перед выполнением должны быть превращены в последовательность таких команд, которые обычно не сложнее чем:

- сложить 2 числа;
- проверить, не является ли число нулем;
- скопировать кусок данных из одной части памяти компьютера в другую.

Эти примитивные команды в совокупности составляют язык, на котором люди могут общаться с компьютером. Такой язык называется **машинным языком**. Разработчик при создании нового компьютера должен решать, какие команды включить в машинный язык этого компьютера. Это зависит от назначения компьютера, от того, какие задачи он должен выполнять. Обычно стараются сделать машинные команды как можно проще, чтобы избежать сложностей при конструировании компьютера и снизить затраты на необходимую электронику. Так как большинство машинных языков очень примитивны, использовать их трудно и утомительно.

Это простое наблюдение с течением времени привело к построению ряда уровней абстракций, каждая из которых надстраивается над абстракцией более низкого уровня. Именно таким образом можно преодолеть сложности при общении с компьютером. Мы называем этот подход **многоуровневой компьютерной организацией**. Так мы и назвали эту книгу. В следующем разделе мы расскажем, что понимаем под этим термином. Затем мы расскажем об истории развития этой проблемы и положении дел в настоящий момент, а также рассмотрим некоторые важные примеры.

Многоуровневая компьютерная организация

Как мы уже сказали, существует огромная разница между тем, что удобно для людей, и тем, что удобно для компьютеров. Люди хотят сделать X, но компьютеры могут сделать только Y. Из-за этого возникают проблемы. Цель данной книги — объяснить, как можно решать эти проблемы.

Языки, уровни и виртуальные машины

Проблему можно решить двумя способами. Оба эти способа включают в себя разработку новых команд, которые более удобны для человека, чем встроенные машинные команды. Эти новые команды в совокупности формируют язык, который мы будем называть Я 1. Встроенные машинные команды тоже формируют язык, и мы будем называть его Я 0. Компьютер может выполнять только программы, написанные на его машинном языке Я 0. Упомянутые два способа решения проблемы различаются тем, каким образом компьютер будет выполнять программы, написанные на языке Я 1.

Первый способ выполнения программы, написанной на языке Я 1, — замена каждой команды на эквивалентный набор команд в языке Я 0. В этом случае компьютер выполняет новую программу, написанную на языке Я 0, вместо старой программы, написанной на Я 1. Эта технология называется **трансляцией**.

Второй способ — написание программы на языке Я 0, которая берет программы, написанные на языке Я 1, в качестве входных данных, рассматривает каждую команду по очереди и сразу выполняет эквивалентный набор команд языка Я 0. Эта технология не требует составления новой программы на Я 0. Она называется **интерпретацией**, а программа, которая осуществляет интерпретацию, называется **интерпретатором**.

Трансляция и интерпретация сходны. При применении обоих методов компьютер в конечном итоге выполняет набор команд на языке Я 0, эквивалентных командам Я 1. Различие лишь в том, что при трансляции вся программа Я 1 переделывается в программу Я 0, программа Я 1 отбрасывается, а новая программа на Я 0 загружается в память компьютера и затем выполняется.

При интерпретации каждая команда программы на Я 1 перекодируется в Я 0 и сразу же выполняется. В отличие от трансляции, здесь не создается новая программа на Я 0, а происходит последовательная перекодировка и выполнение команд. Оба эти метода, а также их комбинация широко используются.

Обычно гораздо проще представить себе существование гипотетического компьютера или **виртуальной машины**, для которой машинным языком является язык Я 1, чем думать о трансляции и интерпретации. Назовем такую виртуальную машину М 1, а виртуальную машину с языком Я 0 — М 0. Если бы такую машину М 1 можно было бы сконструировать без больших денежных затрат, язык Я 0, да и машина, которая выполняет программы на языке Я 0, были бы не нужны. Можно было бы просто писать программы на языке Я 1, а компьютер сразу бы их выполнял. Даже если виртуальная машина слишком дорога или ее очень трудно сконструировать, люди все же могут писать программы для нее. Эти программы могут транслироваться или интерпретироваться программой, написанной на языке Я 0, которая сама могла бы выполняться фактически существующим компьютером. Другими словами, можно писать программы для виртуальных машин, как будто они действительно существуют.

Чтобы трансляция и интерпретация были целесообразными, языки Я 0 и Я 1 не должны сильно различаться. Это значит, что язык Я 1 хотя и лучше, чем Я 0, но все же далек от идеала. Возможно, это несколько обескураживает в свете первоначальной цели создания языка Я 1 — освободить программиста от бремени написа-

ния программ на языке, который более удобен для компьютера, чем для человека. Однако ситуация не так безнадежна.

Очевидное решение этой проблемы — создание еще одного набора команд, которые в большей степени ориентированы на человека и в меньшей степени на компьютер, чем Я 1. Этот третий набор команд также формирует язык, который мы будем называть Я 2, а соответствующую виртуальную машину — М 2. Человек может писать программы на языке Я 2, как будто виртуальная машина с машинным языком Я 2 действительно существует. Такие программы могут или транслироваться на язык Я 1, или выполняться интерпретатором, написанным на языке Я 1.

Изобретение целого ряда языков, каждый из которых более удобен для человека, чем предыдущий, может продолжаться до тех пор, пока мы не дойдем до подходящего нам языка. Каждый такой язык использует своего предшественника как основу, поэтому мы можем рассматривать компьютер в виде ряда **уровней**, как показано на рис. 1.1. Язык, находящийся в самом низу иерархической структуры — самый примитивный, а находящийся на самом верху — самый сложный.



Рис. 1.1. Многоуровневая машина

Между языком и виртуальной машиной существует важная зависимость. У каждой машины есть какой-то определенный машинный язык, состоящий из всех команд, которые эта машина может выполнять. В сущности, машина определяет язык. Сходным образом язык определяет машину, которая может выполнять все программы, написанные на этом языке. Машину, задающуюся определенным языком, очень сложно и дорого сконструировать из электронных схем, но мы можем представить себе такую машину. Компьютер с машинным языком C++ или COBOL был бы слишком сложным, но его можно было бы сконструировать, если учиты-

вать высокий уровень современных технологий. Однако существуют веские причины не создавать такой компьютер: это слишком сложно по сравнению с другими техническими приемами.

Компьютер с n уровнями можно рассматривать как n разных виртуальных машин, у каждой из которых есть свой машинный язык. Термины «уровень» и «виртуальная машина» мы будем использовать как синонимы. Только программы, написанные на Я 0, могут выполняться компьютером без применения трансляции и интерпретации. Программы, написанные на Я 1, Я 2, ..., Я n, должны проходить через интерпретатор более низкого уровня или транслироваться на язык, соответствующий более низкому уровню.

Человеку, который пишет программы для виртуальной машины уровня n, не обязательно знать о трансляторах и интерпретаторах более низких уровней. Машина выполнит эти программы, и не важно, будут ли они выполняться шаг за шагом интерпретатором или их будет выполнять сама машина. В обоих случаях результат один и тот же: программа будет выполнена.

Большинство программистов, использующих машину уровня n, интересуются только самым верхним уровнем, то есть уровнем, который меньше всего сходен с машинным языком. Однако те, кто хочет понять, как в действительности работает компьютер, должны изучить все уровни. Те, кто проектирует новые компьютеры или новые уровни (то есть новые виртуальные машины), также должны быть знакомы со всеми уровнями. Понятия и технические приемы конструирования машин как системы уровней, а также детали уровней составляют главный предмет этой книги.

Современные многоуровневые машины

Большинство современных компьютеров состоит из двух и более уровней. Существуют машины даже с шестью уровнями (рис. 1.2). Уровень 0 — аппаратное обеспечение машины. Его электронные схемы выполняют программы, написанные на языке уровня 1. Ради полноты нужно упомянуть о существовании еще одного уровня, расположенного ниже уровня 0. Этот уровень не показан на рис. 1.2, так как он попадает в сферу электронной техники и, следовательно, не рассматривается в этой книге. Он называется **уровнем физических устройств**. На этом уровне находятся транзисторы, которые являются примитивами для разработчиков компьютеров. Объяснять, как работают транзисторы, — задача физики.

На самом нижнем уровне, **цифровом логическом уровне**, объекты называются **вентилями**. Хотя вентили состоят из аналоговых компонентов, таких как транзисторы, они могут быть точно смоделированы как цифровые средства. У каждого вентиля есть одно или несколько цифровых входных данных (сигналов, представляющих 0 или 1). Вентиль вычисляет простые функции этих сигналов, такие как И или ИЛИ. Каждый вентиль формируется из нескольких транзисторов. Несколько вентилей формируют 1 бит памяти, который может содержать 0 или 1. Биты памяти, объединенные в группы, например, по 16, 32 или 64, формируют **регистры**. Каждый регистр может содержать одно двоичное число до определенного предела. Из вентилей также может состоять сам компьютер. Подробно вентили и цифровой логический уровень мы рассмотрим в главе 3.



Рис. 1.2. Компьютер с шестью уровнями. Способ поддержки каждого уровня указан под ним. В скобках указывается название поддерживающей программы

Следующий уровень — **микроархитектурный уровень**. На этом уровне можно видеть совокупности 8 или 32 регистров, которые формируют локальную память и схему, называемую **АЛУ (арифметико-логическое устройство)**. АЛУ выполняет простые арифметические операции. Регистры вместе с АЛУ формируют **тракт данных**, по которому поступают данные. Основная операция тракта данных состоит в следующем. Выбирается один или два регистра, АЛУ производит над ними какую-либо операцию, например сложения, а результат помещается в один из этих регистров.

На некоторых машинах работа тракта данных контролируется особой программой, которая называется **микропрограммой**. На других машинах тракт данных контролируется аппаратными средствами. В предыдущих изданиях книги мы назвали этот уровень «уровнем микропрограммирования», потому что раньше он почти всегда был интерпретатором программного обеспечения. Поскольку сейчас тракт данных обычно контролируется аппаратным обеспечением, мы изменили название, чтобы точнее отразить смысл.

На машинах, где тракт данных контролируется программным обеспечением, микропрограмма — это интерпретатор для команд на уровне 2. Микропрограмма вызывает команды из памяти и выполняет их одну за другой, используя при этом тракт данных. Например, для того чтобы выполнить команду `ADD`, эта команда вызывается из памяти, ее операнды помещаются в регистры, АЛУ вычисляет сумму, а затем результат переправляется обратно. На компьютере с аппаратным контролем тракта данных происходит такая же процедура, но при этом нет программы, которая контролирует интерпретацию команд уровня 2.

Второй уровень мы будем называть **уровнем архитектуры системы команд**. Каждый производитель публикует руководство для компьютеров, которые он продает, под названием «Руководство по машинному языку» или «Принципы работы компьютера Western Wombat Model 100X» и т. п. Такие руководства содержат информацию именно об этом уровне. Когда они описывают набор машинных команд, они в действительности описывают команды, которые выполняются микропрограммой-интерпретатором или аппаратным обеспечением. Если производитель поставляет два интерпретатора для одной машины, он должен издать два руководства по машинному языку, отдельно для каждого интерпретатора.

Следующий уровень обычно гибридный. Большинство команд в его языке есть также и на уровне архитектуры системы команд (команды, имеющиеся на одном из уровней, вполне могут находиться на других уровнях). У этого уровня есть некоторые дополнительные особенности: набор новых команд, другая организация памяти, способность выполнять две и более программ одновременно и некоторые другие. При построении третьего уровня возможно больше вариантов, чем при построении первого и второго.

Новые средства, появившиеся на третьем уровне, выполняются интерпретатором, который работает на втором уровне. Этот интерпретатор был когда-то назван операционной системой. Команды третьего уровня, идентичные командам второго уровня, выполняются микропрограммой или аппаратным обеспечением, но не операционной системой. Иными словами, одна часть команд третьего уровня интерпретируется операционной системой, а другая часть — микропрограммой. Вот почему этот уровень считается гибридным. Мы будем называть этот уровень **уровнем операционной системы**.

Между третьим и четвертым уровнями есть существенная разница. Нижние три уровня конструируются не для того, чтобы с ними работал обычный программист. Они изначально предназначены для работы интерпретаторов и трансляторов, поддерживающих более высокие уровни. Эти трансляторы и интерпретаторы составляются так называемыми **системными программистами**, которые специализируются на разработке и построении новых виртуальных машин. Уровни с четвертого и выше предназначены для прикладных программистов, решающих конкретные задачи.

Еще одно изменение, появившееся на уровне 4, — способ, которым поддерживаются более высокие уровни. Уровни 2 и 3 обычно интерпретируются, а уровни 4, 5 и выше обычно, хотя и не всегда, поддерживаются транслятором.

Другое различие между уровнями 1, 2, 3 и уровнями 4, 5 и выше — особенность языка. Машинные языки уровней 1, 2 и 3 — цифровые. Программы, написанные на этих языках, состоят из длинных рядов цифр, которые удобны для компьютеров, но совершенно неудобны для людей. Начиная с четвертого уровня, языки содержат слова и сокращения, понятные человеку.

Четвертый уровень представляет собой символическую форму одного из языков более низкого уровня. На этом уровне можно писать программы в приемлемой для человека форме. Эти программы сначала транслируются на язык уровня 1, 2 или 3, а затем интерпретируются соответствующей виртуальной или фактически существующей машиной. Программа, которая выполняет трансляцию, называется **ассемблером**.

Пятый уровень обычно состоит из языков, разработанных для прикладных программистов. Такие языки называются **языками высокого уровня**. Существуют сотни языков высокого уровня. Наиболее известные среди них — BASIC, C, C++, Java, LISP и Prolog. Программы, написанные на этих языках, обычно транслируются на уровень 3 или 4. Трансляторы, которые обрабатывают эти программы, называются **компиляторами.** Отметим, что иногда также используется метод интерпретации. Например, программы на языке Java обычно интерпретируются.

В некоторых случаях пятый уровень состоит из интерпретатора для такой сферы приложения, как символическая математика. Он обеспечивает данные и операции для решения задач в этой сфере в терминах, понятных людям, сведущим в символической математике.

Вывод: компьютер проектируется как иерархическая структура уровней, каждый из которых надстраивается над предыдущим. Каждый уровень представляет собой определенную абстракцию с различными объектами и операциями. Рассматривая компьютер подобным образом, мы можем не принимать во внимание ненужные нам детали и свести сложный предмет к более простому для понимания.

Набор типов данных, операций и особенностей каждого уровня называется **архитектурой**. Архитектура связана с аспектами, которые видны программисту. Например, сведения о том, сколько памяти можно использовать при написании программы, — часть архитектуры. А аспекты разработки (например, какая технология используется при создании памяти) не являются частью архитектуры. Изучение того, как разрабатываются те части компьютерной системы, которые видны программистам, называется изучением **компьютерной архитектуры**. Термины «компьютерная архитектура» и «компьютерная организация» означают в сущности одно и то же.

Развитие многоуровневых машин

В этом разделе мы кратко изложим историю развития многоуровневых машин, покажем, как число и природа уровней менялись с годами. Программы, написанные на машинном языке (уровень 1), могут сразу выполняться электронными схемами компьютера (уровень 0), без применения интерпретаторов и трансляторов. Эти электронные схемы вместе с памятью и средствами ввода-вывода формируют **аппаратное обеспечение**. Аппаратное обеспечение состоит из осязаемых объектов — интегральных схем, печатных плат, кабелей, источников электропитания, запоминающих устройств и принтеров. Абстрактные понятия, алгоритмы и команды не относятся к аппаратному обеспечению.

Программное обеспечение, напротив, состоит из алгоритмов (подробных последовательностей команд, которые описывают, как решить задачу) и их компьютерных представлений, то есть программ. Программы могут храниться на жестком диске, гибком диске, компакт-диске или других носителях, но в сущности программное обеспечение — это набор команд, составляющих программы, а не физические носители, на которых эти программы записаны.

В самых первых компьютерах граница между аппаратным и программным обеспечением была очевидна. Со временем, однако, произошло значительное размывание этой границы, в первую очередь благодаря тому, что в процессе развития

компьютеров уровни добавлялись, убирались и сливались. В настоящее время очень сложно отделить их друг от друга. В действительности центральная тема этой книги может быть выражена так: *аппаратное и программное обеспечение логически эквивалентны.*

Любая операция, выполняемая программным обеспечением, может быть встроена в аппаратное обеспечение (желательно после того, как она осознана). Карен Панетта Ленц говорил: «Аппаратное обеспечение — это всего лишь окаменевшее программное обеспечение». Конечно, обратное тоже верно: любая команда, выполняемая аппаратным обеспечением, может быть смоделирована в программном обеспечении. Решение разделить функции аппаратного и программного обеспечения основано на таких факторах, как стоимость, скорость, надежность, а также частота ожидаемых изменений. Существует несколько жестких правил, сводящихся к тому, что X должен быть в аппаратном обеспечении, а Y должен программироваться. Эти решения изменяются в зависимости от тенденций в развитии компьютерных технологий.

Изобретение микропрограммирования

У первых цифровых компьютеров в 1940-х годах было только 2 уровня: уровень архитектуры набора команд, на котором осуществлялось программирование, и цифровой логический уровень, который выполнял программы. Схемы цифрового логического уровня были сложны для производства и понимания и ненадежны.

В 1951 году Морис Уилкс, исследователь Кембриджского университета, предложил идею разработки трехуровневого компьютера для того, чтобы упростить аппаратное обеспечение [158]. Эта машина должна была иметь встроенный неизменяемый интерпретатор (микропрограмму), функция которого заключалась в выполнении программ посредством интерпретации. Так как аппаратное обеспечение должно было теперь вместо программ уровня архитектуры команд выполнять только микропрограммы с ограниченным набором команд, требовалось меньшее количество электронных схем. Поскольку электронные схемы тогда делались из электронных ламп, такое упрощение должно было сократить количество ламп и, следовательно, увеличить надежность.

В 50-е годы было построено несколько трехуровневых машин. В 60-х годах число таких машин значительно увеличилось. К 70-м годам идея о том, что написанная программа сначала должна интерпретироваться микропрограммой, а не выполняться непосредственно электроникой, стала преобладающей.

Изобретение операционной системы

В те времена, когда компьютеры только появились, принципы работы с ними сильно отличались от современных. Одним компьютером пользовалось большое количество людей. Рядом с машиной лежал листок бумаги, и если программист хотел запустить свою программу, он записывался на какое-то определенное время, скажем, на среду с 3 часов ночи до 5 утра (многие программисты любили работать в тишине). В назначенное время программист направлялся в комнату, где стояла машина, с пачкой перфокарт, которые тогда служили средством ввода, и карандашом. Каждая перфокарта содержала 80 колонок; на ней в определенных местах

пробивались отверстия. Войдя в комнату, программист вежливо просил предыдущего программиста освободить место и приступал к работе.

Если он хотел запустить программу на языке FORTRAN, ему необходимо было пройти следующие этапы:

1. Он подходил к шкафу, где находилась библиотека программ, брал большую зеленую стопку перфокарт с надписью «Компилятор FORTRAN», помещал их в считывающее устройство и нажимал кнопку «Пуск».
2. Затем он помещал стопку карточек со своей программой, написанной на языке FORTRAN, в считывающее устройство и нажимал кнопку «Продолжить». Программа считывалась.
3. Когда компьютер прекращал работу, программист считывал свою программу во второй раз. Некоторые компиляторы требовали только одного считывания перфокарт, но в большинстве случаев необходимо было производить эту процедуру несколько раз. Каждый раз нужно было считывать большую стопку перфокарт.
4. В конце концов трансляция завершалась. Программист часто становился очень нервным, потому что если компилятор находил ошибку в программе, ему приходилось исправлять ее и начинать процесс ввода программы заново. Если ошибок не было, компилятор выдавал программу на машинном языке на перфокартах.
5. Тогда программист помещал эту программу на машинном языке в устройство считывания вместе с пачкой перфокарт из библиотеки подпрограмм и загружал обе эти программы.
6. Начиналось выполнение программы. В большинстве случаев она не работала или неожиданно останавливалась в середине. Обычно в этом случае программист начинал дергать переключатели на пульте и смотрел на лампочки. В случае удачи он находил и исправлял ошибку, подходил к шкафу, в котором лежал большой зеленый компилятор FORTRAN, и начинал все заново. В случае неудачи он делал распечатку содержания памяти, что называлось **разгрузкой оперативного запоминающего устройства**, и брал эту распечатку домой для изучения.

Это процедура была обычной на протяжении многих лет. Программистам приходилось учиться, как работать с машиной и что нужно делать, если она выходила из строя, что происходило довольно часто. Машина постоянно простаивала без работы, пока люди носили перфокарты по комнате или ломали головы над тем, почему программа не работает.

В 60-е годы человек попытался сократить количество потерянного времени, автоматизировав работу оператора. Программа под названием **«операционная система»** теперь содержалась в компьютере все время. Программист приносил пачку перфокарт со специальной программой, которая выполнялась операционной системой. На рисунке 1.3 показана модель пачки перфокарт для первой широко распространенной операционной системы FMS (FORTRAN Monitor System) к компьютеру IBM-709.



Рис. 1.3. Схема работы с операционной системой FMS

Операционная система считывала перфокарту *JOB и использовала содержащуюся на ней информацию для учета системных ресурсов (звездочка ставилась, чтобы отличать перфокарты с программой контроля от перфокарт с данными). Затем операционная система считывала перфокарту *FORTRAN, которая представляла собой команду для загрузки компилятора FORTRAN с магнитной ленты. После этого компилятор считывал и компилировал программу, написанную на языке FORTRAN. Как только компилятор заканчивал работу, операционная система считывала перфокарту *DATA — команду по выполнению транслированной программы с использованием перфокарт данных.

Операционная система была придумана для того, чтобы автоматизировать работу оператора (отсюда и название), но это не единственное ее назначение. Создание операционной системы было первым шагом в развитии новой виртуальной машины. Перфокарту *FORTRAN можно рассматривать как виртуальную команду к компилятору, а перфокарту *DATA — как виртуальную команду для выполнения программы. И хотя этот уровень состоял всего из двух команд, он был первым этапом в развитии виртуальных машин.

В последующие годы операционные системы все больше и больше усложнялись. К уровню архитектуры команд добавлялись новые команды, приспособления и особенности, и в итоге сформировался новый уровень. Некоторые команды нового уровня были идентичны командам предыдущего, но другие (в частности команды ввода-вывода) полностью отличались. Эти новые команды тогда назывались **макросами операционной системы** или **вызовами супервизора**. Сейчас обычно используется термин **«системный вызов»**.

Первые операционные системы считывали пачки перфокарт и распечатывали результат на принтере. Такая организация вычислений называлась **пакетным**

режимом. Чтобы получить результат, обычно нужно было ждать несколько часов. При таких условиях было трудно развивать программное обеспечение.

В начале 60-х годов исследователи Массачусетского технологического института (МТИ) разработали операционную систему, которая давала возможность работать с компьютером сразу нескольким программистам. В этой системе к центральному компьютеру через телефонные линии подсоединялись отдаленные терминалы. Таким образом, центральный процессор разделялся между большим количеством пользователей. Программист мог напечатать свою программу и получить результаты почти сразу прямо в офисе, гараже или где бы то ни было еще (там, где находился терминал). Эти системы назывались (и сейчас называются) **системами с разделением времени**.

Хотя нас интересуют только те части операционной системы, которые интерпретируют команды третьего уровня, необходимо понимать, что это не единственная функция операционных систем.

Перемещение функциональности системы на уровень микрокода

С 1970 года, когда микропрограммирование стало обычным, у производителей появилась возможность вводить новые машинные команды путем расширения микропрограммы, то есть с помощью программирования. Это открытие привело к виртуальному взрыву в производстве программ машинных команд, поскольку производители начали конкурировать друг с другом, стараясь выпустить лучшие программы. Эти команды не представляли особой ценности, поскольку те же задачи можно было легко решить, используя уже существующие программы, но обычно они работали немного быстрее. Например, во многих компьютерах использовалась команда INC (INCrement), которая прибавляла к числу единицу. Тогда уже существовала общая команда сложения ADD, и не было необходимости вводить новую команду, прибавляющую к числу единицу. Тем не менее команда INC работала немного быстрее, чем команда ADD, поэтому ее также включили в набор команд.

Многие программы были добавлены в микропрограмму по той же причине. Среди них можно назвать:

1. Команды для умножения и деления целых чисел.
2. Команды для арифметических действий над числами с плавающей точкой.
3. Команды для вызова и прекращения действия процедур.
4. Команды для ускорения циклов.
5. Команды для работы со строкой символов.

Как только производители поняли, что добавлять новые команды очень легко, они начали думать, какие дополнительные технические характеристики можно добавить к микропрограмме. Приведем несколько примеров:

1. Ускорение работы с массивами (индексная и косвенная адресация).
2. Перемещение программы из одного раздела памяти в другой после запуска программы (настройка).

3. Системы прерывания, которые дают сигнал процессору, как только закончена операция ввода или вывода.
4. Способность приостановить одну программу и начать другую, используя небольшое число команд (переключение процесса).

В дальнейшем дополнительные команды и технические характеристики вводились также для ускорения работы компьютеров.

Устранение микропрограммирования

В 60-х–70-х годах количество микропрограмм сильно увеличилось. Однако они работали все медленнее и медленнее, поскольку требовали большого объема памяти. В конце концов исследователи осознали, что с устранением микропрограммы резко сократится количество команд и компьютеры станут работать быстрее. Таким образом, компьютеры вернулись к тому состоянию, в котором они находились до изобретения микропрограммирования.

Мы рассмотрели развитие компьютеров, чтобы показать, что граница между аппаратным и программным обеспечением постоянно перемещается. Сегодняшнее программное обеспечение может быть завтрашним аппаратным обеспечением и наоборот. Так же обстоит дело и с уровнями — между ними нет четких границ. Для программиста не важно, как на самом деле выполняется команда (за исключением, может быть, скорости выполнения). Программист, работающий на уровне архитектуры системы, может использовать команду умножения, как будто это команда аппаратного обеспечения, и даже не задумываться об этом. То, что для одного человека — программное обеспечение, для другого — аппаратное. Мы еще вернемся к этим вопросам ниже.

Развитие компьютерной архитектуры

В период развития компьютерных технологий были разработаны сотни разных компьютеров. Многие из них давно забыты, но некоторые сильно повлияли на современные идеи. В этом разделе мы дадим краткий обзор некоторых ключевых исторических моментов, чтобы лучше понять, каким образом разработчики дошли до создания современных компьютеров. Мы рассмотрим только основные моменты развития, оставив многие подробности за скобками.

Компьютеры, которые мы будем рассматривать, представлены в табл. 1.1.

Нулевое поколение — механические компьютеры (1642–1945)

Первым человеком, создавшим счетную машину, был французский ученый Блез Паскаль (1623–1662), в честь которого назван один из языков программирования. Паскаль сконструировал эту машину в 1642 году, когда ему было всего 19 лет, для своего отца, сборщика налогов. Она была механическая: с шестеренками и ручным приводом. Счетная машина Паскаля могла выполнять только операции сложения и вычитания.

Тридцать лет спустя великий немецкий математик Готфрид Вильгельм Лейбниц (1646–1716) построил другую механическую машину, которая кроме сложения и вычитания могла выполнять операции умножения и деления. В сущности, Лейбниц три века назад создал подобие карманного калькулятора с четырьмя функциями.

Еще через 150 лет профессор математики Кембриджского университета Чарльз Бэббидж (1792–1871), изобретатель спидометра, разработал и сконструировал **разностную машину**. Эта механическая машина, которая, как и машина Паскаля, могла только складывать и вычитать, подсчитывала таблицы чисел для морской навигации. В машину был заложен только один алгоритм — метод конечных разностей с использованием полиномов. У этой машины был довольно интересный способ вывода информации: результаты выдавливались стальным штампом на медной дощечке, что предвосхитило более поздние средства ввода-вывода — перфокарты и компакт-диски.

Хотя это устройство работало довольно неплохо, Бэббиджу вскоре наскучила машина, выполнявшая только один алгоритм. Он потратил очень много времени, большую часть своего семейного состояния и еще 17000 фунтов, выделенных правительством, на разработку **аналитической машины**. У аналитической машины было 4 компонента: запоминающее устройство (память), вычислительное устройство, устройство ввода (для считывания перфокарт), устройство вывода (перфоратор и печатающее устройство). Память состояла из 1000 слов по 50 десятичных разрядов, каждое из которых содержало переменные и результаты. Вычислительное устройство принимало операнды из памяти, затем выполняло операции сложения, вычитания, умножения или деления и возвращало полученный результат обратно в память. Как и разностная машина, это устройство было механическим.

Преимущество аналитической машины заключалось в том, что она могла выполнять разные задачи. Она считывала команды с перфокарт и выполняла их. Некоторые команды приказывали машине взять 2 числа из памяти, перенести их в вычислительное устройство, произвести над ними операцию (например, сложить) и отправить результат обратно в запоминающее устройство. Другие команды проверяли число, а иногда совершали операцию перехода в зависимости от того, положительное оно или отрицательное. Если в считывающее устройство вводились перфокарты с другой программой, то машина выполняла другой набор операций. А разностная машина могла осуществлять только один алгоритм.

Поскольку эта аналитическая машина программировалась на ассемблере, ей было необходимо программное обеспечение. Чтобы создать это программное обеспечение, Бэббидж нанял молодую женщину — Аду Августу Ловлейс, дочь знаменитого британского поэта Байрона. Ада Ловлейс была первым в мире программистом. В ее честь назван современный язык программирования Ada.

К несчастью, Бэббидж никогда не отлаживал компьютер. Ему нужны были тысячи и тысячи шестеренок, сделанных с такой точностью, которая была невозможна в XIX веке. Но идеи Бэббиджа опередили его эпоху, и даже сегодня большинство современных компьютеров по строению сходны с аналитической машиной. Поэтому справедливо будет сказать, что Бэббидж был дедушкой современного цифрового компьютера.

Таблица 1.1. Основные этапы развития компьютеров

Год выпуска	Название компьютера	Создатель	Примечания
1834	Аналитическая машина	Бэббидж	Первая попытка построить цифровой компьютер
1936	Z1	Зус	Первая релейная вычислительная машина
1943	COLOSSUS	Британское правительство	Первый электронный компьютер
1944	Mark I	Айкен	Первый американский многоцелевой компьютер
1946	ENIAC I	Экерт/ Моушли	Отсюда начинается история современных компьютеров
1949	EDSAC	Уилкс	Первый компьютер с программами, хранящимися в памяти
1951	Whirlwind I	МТИ	Первый компьютер реального времени
1952	IAS	Фон Нейман	Этот проект используется в большинстве современных компьютеров
1960	PDP-1	DEC	Первый мини-компьютер (продано 50 экземпляров)
1961	1401	IBM	Очень популярный маленький компьютер
1962	7094	IBM	Очень популярная небольшая вычислительная машина
1963	B5000	Burroughs	Первая машина, разработанная для языка высокого уровня
1964	360	IBM	Первое семейство компьютеров
1964	6600	CDC	Первый суперкомпьютер для научных расчетов
1965	PDP-8	DEC	Первый мини-компьютер массового потребления (продано 50 000 экземпляров)
1970	PDP-11	DEC	Эти мини-компьютеры доминировали на компьютерном рынке в 70-е годы.
1974	8080	Intel	Первый универсальный 8-битный компьютер на микросхеме
1974	CRAY-1	Cray	Первый векторный супер-компьютер
1978	VAX	DEC	Первый 32-битный суперминикомпьютер
1981	IBM PC	IBM	Началась эра современных персональных компьютеров
1985	MIPS	MIPS	Первый компьютер RISC
1987	SPARC	Sun	Первая рабочая станция RISC на основе процессора SPARC
1990	RS6000	IBM	Первый суперскалярный компьютер

В конце 30-х годов XX века немец Конрад Зус сконструировал несколько автоматических счетных машин с использованием электромагнитных реле. Ему не удалось получить денежные средства от правительства на свои разработки, потому что началась война. Зус ничего не знал о работе Бэббиджа, и его машины были уничтожены во время бомбежки Берлина в 1944 году, поэтому его работа никак не повлияла на будущее развитие компьютерной техники. Однако он был одним из пионеров в этой области.

Немного позже счетные машины были сконструированы в Америке. Машина Атанасова была чрезвычайно развитой для того времени. В ней использовалась бинарная арифметика и информационные емкости, которые периодически обновлялись, чтобы избежать уничтожения данных. Современная динамическая память (ОЗУ) работает точно по такому же принципу. К несчастью, эта машина так и не стала действующей. В каком-то смысле Атанасов был похож на Бэббиджа: мечтатель, которого не устраивали технологии своего времени.

Компьютер Стибитса действительно работал, хотя и был примитивнее, чем машина Атанасова. Стибитс продемонстрировал работу своей машины на конференции в Дартмутском колледже в 1940 году. На этой конференции присутствовал Джон Моушли, ничем не знаменитый профессор физики из университета Пенсильвании. Позднее он стал очень известным в области компьютерных разработок.

Пока Зус, Стибитс и Атанасов разрабатывали автоматические счетные машины, молодой Говард Айкен с трудом проектировал ручные счетные машины как часть своего философского исследования в Гарварде. После окончания исследования Айкен осознал важность автоматических вычислений. Он пошел в библиотеку, узнал о работе Бэббиджа и решил создать из реле такой же компьютер, который Бэббиджу не удалось создать из зубчатых колес.

Работа над первым компьютером Айкена «Mark I» была закончена в 1944 году. Компьютер содержал 72 слова по 23 десятичных разряда каждое и мог выполнить любую команду за 6 секунд. На устройствах ввода-вывода использовалась перфолента. К тому времени, как Айкен закончил работу над компьютером «Mark II», релейные компьютеры уже устарели. Началась эра электроники.

Первое поколение — электронные лампы (1945–1955)

Стимулом к созданию электронного компьютера стала Вторая мировая война. В начале войны германские подводные лодки разрушали британские корабли. Германские адмиралы посылали на подводные лодки по радио команды, а англичане могли перехватывать эти команды. Проблема заключалась в том, что эти радиопослания были закодированы с помощью прибора под названием **ENIGMA**, предшественник которого был спроектирован изобретателем-дилетантом и бывшим президентом США Томасом Джефферсоном.

В начале войны англичанам удалось приобрести ENIGMA у поляков, которые, в свою очередь, украли его у немцев. Однако чтобы расшифровать закодированное послание, требовалось огромное количество вычислений, и их нужно было произвести сразу после того, как радиопослание было перехвачено. Поэтому британское правительство основало секретную лабораторию для создания электронного компьютера под названием **COLOSSUS**. В создании этой машины принимал участие знаменитый британский математик Алан Тьюринг. COLOSSUS работал уже в 1943 году, но так как британское правительство полностью контролировало этот проект и рассматривало его как военную тайну на протяжении 30 лет, COLOSSUS не мог служить основой дальнейшего развития компьютеров. Мы упомянули его только потому, что это был первый в мире электронный цифровой компьютер.

Вторая мировая война повлияла и на развитие компьютерной техники в США. Армии нужны были таблицы стрельбы, которые использовались при нацеливании тяжелой артиллерии. Сотни женщин нанимались для высчитывания этих таблиц на ручных счетных машинах (считалось, что женщины более аккуратны при расчетах, чем мужчины). Тем не менее этот процесс требовал много времени, и часто случались ошибки.

Джон Моушли, который был знаком с работами Атанасова и Стибитса, понимал, что армия заинтересована в создании механических счетных машин. Он потребовал от армии финансирования работ по созданию электронного компьютера. Требование было удовлетворено в 1943 году, и Моушли со своим студентом, Дж. Преспером Экертом, начали конструировать электронный компьютер, который они назвали **ENIAC** (Electronic Numerical Integrator and Computer — электронный цифровой интегратор и калькулятор). Он состоял из 18 000 электровакуумных ламп и 1500 реле. ENIAC весил 30 тонн и потреблял 140 киловатт электроэнергии. У машины было 20 регистров, каждый из которых мог содержать 10-разрядное десятичное число. (Десятичный регистр — это память очень маленького объема, которая может вмещать число до какого-либо определенного максимального количества разрядов, что-то вроде одометра, который запоминает километраж пройденного автомобилем пути.) В ENIAC было установлено 6000 многоканальных переключателей и множество кабелей было протянуто к розеткам.

Работа над машиной была закончена в 1946 году, когда она уже была не нужна.

Но поскольку война закончилась, Моушли и Экерту позволили организовать школу, где они рассказывали о своей работе коллегам-ученым. С этой школы началось развитие интереса к созданию больших цифровых компьютеров.

После появления школы и другие исследователи взялись за конструирование электронных вычислительных машин. Первым рабочим компьютером был EDSAC (1949 год). Эту машину сконструировал Морис Уилкс в Кембриджском университете. Далее JOHNIAC — в корпорации Rand, ILLIAC — в Университете Иллинойса, MANIAC — в лаборатории Лос-Аламоса и WEIZAC — в Институте Вайцмана в Израиле.

Экерт и Моушли вскоре начали работу над машиной **EDVAC** (Electronic Discrete Variable Computer — электронная дискретная параметрическая машина). К несчастью, этот проект закрылся, когда они ушли из университета, чтобы основать компьютерную корпорацию в Филадельфии (Силиконовой долины тогда еще не было). После ряда слияний эта компания превратилась в Unisys Corporation.

Экерт и Моушли хотели получить патент на изобретение цифровой вычислительной машины. После нескольких лет судебной тяжбы было вынесено решение, что патент недействителен, так как цифровую вычислительную машину изобрел Атанасов, хотя он и не запатентовал свое изобретение.

В то время как Экерт и Моушли работали над машиной EDVAC, один из участников проекта ENIAC, Джон фон Нейман, поехал в Институт специальных исследований в Принстоне, чтобы сконструировать свою собственную версию EDVAC, машину **IAS**[1]. Фон Нейман был гением в тех же областях, что и Леонардо да Винчи. Он знал много языков, был специалистом в физике и математике и обладал феноменальной памятью: он помнил все, что когда-либо слышал, видел или читал.

[1] Сокр. от Immediate Address Storadge — память с прямой адресацией. — *Примеч. перев.*

Он мог дословно процитировать по памяти тексты книг, которые читал несколько лет назад. Когда фон Нейман стал интересоваться вычислительными машинами, он уже был самым знаменитым математиком в мире.

Фон Нейман вскоре осознал, что создание компьютеров с большим количеством переключателей и кабелей требует длительного времени и очень утомительно. Он пришел к мысли, что программа должна быть представлена в памяти компьютера в цифровой форме, вместе с данными. Он также отметил, что десятичная арифметика, используемая в машине ENIAC, где каждый разряд представлялся 10 электронными лампами (1 включена и 9 выключены), должна быть заменена бинарной арифметикой.

Основной проект, который он описал вначале, известен сейчас как **фон-неймановская вычислительная машина**. Он был использован в EDSAC, первой машине с программой в памяти, и даже сейчас, более чем полвека спустя, является основой большинства современных цифровых компьютеров. Этот замысел и машина IAS оказали очень большое влияние на дальнейшее развитие компьютерной техники, поэтому стоит кратко описать его. Схема архитектуры этой машины дана на рис. 1.4.

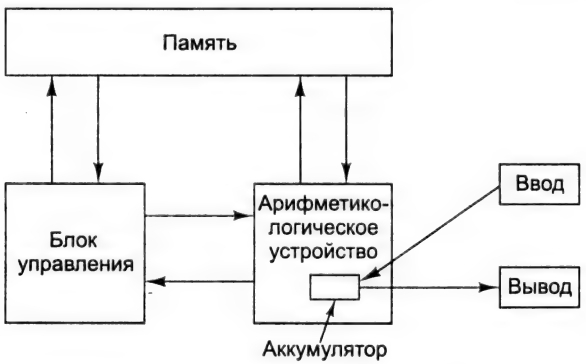


Рис. 1.4. Схема фон-неймановской вычислительной машины

Машина фон Неймана состояла из пяти основных частей: памяти, арифметико-логического устройства, устройства управления, а также устройств ввода-вывода. Память включала 4096 слов, каждое слово содержало 40 битов, бит — это 0 или 1. Каждое слово содержало или 2 команды по 20 битов, или целое число со знаком на 40 битов. 8 битов указывали на тип команды, а остальные 12 битов определяли одно из 4096 слов.

Внутри арифметико-логического устройства находился особый внутренний регистр в 40 битов, так называемый **аккумулятор**. Типичная команда добавляла слово из памяти к аккумулятору или сохраняла содержимое аккумулятора в памяти. Эта машина не выполняла арифметические операции с плавающей точкой, так как фон Нейман понимал, что любой сведущий математик был способен держать плавающую запятую в голове.

Примерно в то же время, когда фон Нейман работал над машиной IAS, исследователи МТИ разрабатывали свой компьютер Whirlwind I. В отличие от IAS, ENIAC и других машин того же типа со словами большой длины, машина Whirlwind I содержала слова по 16 битов и была предназначена для работы в реальном времени.

Этот проект привел к изобретению памяти на магнитном сердечнике (изобретатель Джей Форрестер), а затем и первого серийного мини-компьютера.

В то время IBM была маленькой компанией, производившей перфокарты и механические машины для их сортировки. Хотя фирма IBM частично финансировала проект Айкена, она не интересовалась компьютерами и только в 1953 году построила компьютер IBM-701, через много лет после того, как компания Экерта и Моушли со своим компьютером UNIVAC стала номером один на компьютерном рынке.

В IBM-701 было 2048 слов по 36 битов, каждое слово содержало две команды. Он стал первым компьютером, лидирующим на рынке в течение десяти лет. Через три года появился IBM-704, у которого было 4 Кбайт памяти на магнитных сердечниках, команды по 36 битов и процессор с плавающей точкой. В 1958 году компания IBM начала работу над последним компьютером на электронных лампах, IBM-709, который по сути представлял собой усложненную версию IBM-704.

Второе поколение — транзисторы (1955–1965)

Транзистор был изобретен сотрудниками лаборатории Bell Laboratories Джоном Бардином, Уолтером Браттейном и Уильямом Шокли, за что в 1956 году они получили Нобелевскую премию в области физики. В течение десяти лет транзисторы произвели революцию в производстве компьютеров, и к концу 50-х годов компьютеры на вакуумных лампах устарели. Первый компьютер на транзисторах был построен в лаборатории МТИ. Он содержал слова из 16 битов, как и Whirlwind I. Компьютер назывался TX-0 (Transistorized eXperimental computer 0 — экспериментальная транзисторная вычислительная машина 0) и предназначался только для тестирования машины TX-2.

Машина TX-2 не имела большого значения, но один из инженеров из этой лаборатории, Кеннет Ольсен, в 1957 году основал компанию DEC (Digital Equipment Corporation — корпорация по производству цифровой аппаратуры), чтобы производить серийную машину, сходную с TX-0. Эта машина, PDP-1, появилась только через четыре года главным образом потому, что капиталисты, финансирующие DEC, считали производство компьютеров невыгодным. Поэтому компания DEC продавала в основном небольшие электронные платы.

PDP-1 появился только в 1961 году. У него было 4 Кбайт слов по 18 битов и время цикла 5 микросекунд. Этот параметр был в два раза меньше, чем у IBM-7090, транзисторного аналога IBM-709. PDP-1 был самым быстрым компьютером в мире в то время. PDP-1 стоил $120000, а IBM-7090 стоил миллионы. Компания DEC продала десятки компьютеров PDP-1, и так появилась компьютерная промышленность.

Одну из первых машин модели PDP-1 отдали в МТИ, где она сразу привлекла внимание некоторых молодых исследователей, подающих большие надежды. Одним из нововведений PDP-1 был дисплей с размером 512 на 512 пикселов, на котором можно было рисовать точки. Вскоре студенты МТИ составили специальную программу для PDP-1, чтобы играть в «Войну миров» — первую в мире компьютерную игру.

Через несколько лет DEC разработал модель PDP-8, 12-битный компьютер. PDP-8 стоил гораздо дешевле, чем PDP-1 ($16000). Главное нововведение — одна шина (Omnibus) (рис. 1.5). **Шина** — это набор параллельно соединенных проводов

для связи компонентов компьютера. Это нововведение сильно отличало PDP-8 от IAS. Такая структура с тех пор стала использоваться во всех компьютерах. Компания DEC продала 50 000 компьютеров модели PDP-8 и стала лидером на рынке мини-компьютеров.

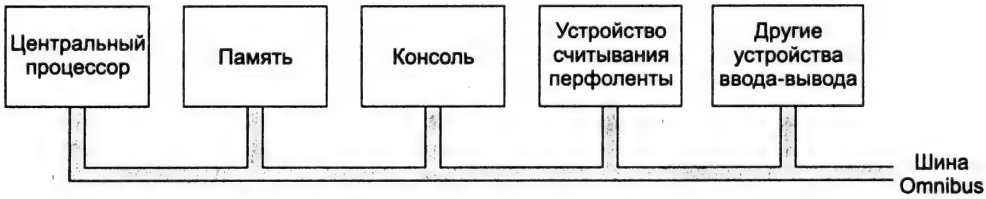


Рис. 1.5. Шина компьютера PDP-8

Как уже было сказано, с изобретением транзисторов компания IBM построила транзисторную версию IBM-709 — IBM-7090, а позднее — IBM-7094. У нее время цикла составляло 2 микросекунды, а память состояла из 32 К слов по 16 битов. IBM-7090 и IBM-7094 были последними компьютерами типа ENIAC, но они широко использовались для научных расчетов в 60-х годах прошлого века.

Компания IBM также выпускала компьютеры IBM-1401 для коммерческих расчетов. Эта машина могла считывать и записывать магнитные ленты и перфокарты и распечатывать результат так же быстро, как и IBM-7094, но при этом стоила дешевле. Для научных вычислений она не подходила, но зато была очень удобна для ведения деловых записей.

У IBM-1401 не было регистров и фиксированной длины слова. Память содержала 4 Кбайт по 8 битов (4 Кбайт). Каждый байт содержал символ в 6 битов, административный бит и бит для указания конца слова. У команды `MOVE`, например, есть исходный адрес и адрес пункта назначения. Эта команда перемещает байты из первого адреса во второй, пока бит конца слова не примет значение 1.

В 1964 году компания CDC (Control Data Corporation) выпустила машину 6600, которая работала почти на порядок быстрее, чем IBM-7094. Этот компьютер для сложных расчетов пользовался большой популярностью, и компания CDC пошла в гору. Секрет столь высокой скорости работы заключался в том, что внутри ЦПУ (центрального процессора) находилась машина с высокой степенью параллелизма. У нее было несколько функциональных устройств для сложения, умножения и деления, и все они могли работать одновременно. Для того чтобы машина быстро работала, нужно было составить хорошую программу, но приложив некоторые усилия, можно было сделать так, чтобы машина выполняла 10 команд одновременно.

Внутри машины 6600 было встроено несколько маленьких компьютеров. ЦПУ, таким образом, производило только подсчет чисел, а остальные функции (управление работой машины, а также ввод и вывод информации) выполняли маленькие компьютеры. Некоторые принципы устройства 6600 используются и в современных компьютерах.

Разработчик компьютера 6600 Сеймур Крей был легендарной личностью, как и фон Нейман. Он посвятил всю свою жизнь созданию очень мощных компьютеров, которые сейчас называют **суперкомпьютерами**. Среди них можно назвать CDC-6600, CDC-7600 и Cray-1. Сеймур Крей также является автором известного

«алгоритма покупки автомобилей»: вы идете в магазин, ближайший к вашему дому, показываете на машину, ближайшую к двери, и говорите: «Я беру эту». Этот алгоритм позволяет тратить минимум времени на не очень важные дела (покупку автомобилей) и оставляет большую часть времени на важные (разработку суперкомпьютеров).

Следует упомянуть еще один компьютер — Burroughs B5000. Разработчики машин PDP-1, IBM-7094 и CDC-6600 занимались только аппаратным обеспечением, стараясь снизить его стоимость (DEC) или заставить работать быстрее (IBM и CDC). Программное обеспечение не менялось. Производители B5000 пошли другим путем. Они разработали машину с намерением программировать ее на языке Algol 60 (предшественнике языка Pascal), сконструировав аппаратное обеспечение так, чтобы упростить задачу компилятора. Так появилась идея, что программное обеспечение также нужно учитывать при разработке компьютера. Но вскоре эта идея была забыта.

Третье поколение — интегральные схемы (1965–1980)

Изобретение кремниевой интегральной схемы в 1958 году (изобретатель — Роберт Нойс) дало возможность помещать десятки транзисторов на одну небольшую микросхему. Компьютеры на интегральных схемах были меньшего размера, работали быстрее и стоили дешевле, чем их предшественники на транзисторах. Ниже описаны наиболее значительные из них.

К 1964 году компания IBM лидировала на компьютерном рынке, но существовала одна большая проблема: компьютеры IBM-7094 и IBM-1401, которые она выпускала, были несовместимы друг с другом. Один из них предназначался для сложных расчетов, в нем использовалась двоичная арифметика на регистрах по 36 битов, а во втором использовалась десятичная система счисления и слова разной длины. У многих покупателей были оба компьютера, и им не нравилось, что они совершенно несовместимы.

Когда пришло время заменить эти две серии компьютеров, компания IBM сделала решительный шаг. Она выпустила серию компьютеров на транзисторах, System/360, которые были предназначены и для научных, и для коммерческих расчетов. System/360 содержала много нововведений. Это было целое семейство компьютеров с одним и тем же языком (ассемблером). Каждая новая модель была больше по размеру и по мощности, чем предыдущая. Компания могла заменить IBM-1401 на IBM-360 (модель 30), а IBM-7094 — на IBM-360 (модель 75). Модель 75 была больше по размеру, работала быстрее и стоила дороже, но программы, написанные для одной из них, могли использоваться для другой. На практике программы, написанные для маленькой модели, выполнялись большой моделью без особых затруднений. Но в случае переноса программного обеспечения с большой машины на маленькую могло не хватить памяти. И все же создание такой серии компьютеров было большим достижением. Идея создания семейств компьютеров вскоре стала очень популярной, и в течение нескольких лет большинство компьютерных компаний выпустило целые серии сходных машин с разной сто-

имостью и функциями. В табл. 1.2 показаны некоторые параметры первых моделей из семейства IBM-360. О других моделях этого семейства мы расскажем ниже.

Таблица 1.2. Первые модели серии IBM-360

Параметры	Модель 30	Модель 40	Модель 50	Модель 60
Относительная производительность	1	3,5	10	21
Время цикла, нс	1000	625	500	250
Максимальный объем памяти, Кбайт	64	256	256	512
Количество байтов, вызываемых из памяти за 1 цикл	1	2	4	16
Максимальное количество каналов данных	3	3	4	6

Еще одно нововведение в IBM-360 — **мультипрограммирование**. В памяти компьютера могло находиться одновременно несколько программ, и пока одна программа ждала, когда закончится процесс ввода-вывода, другая выполнялась.

IBM-360 была первой машиной, которая могла полностью имитировать работу других компьютеров. Маленькие модели могли имитировать IBM-1401, а большие — IBM-7094, поэтому программисты могли оставлять свои старые программы без изменений и использовать их в работе с IBM-360. Некоторые модели IBM-360 выполняли программы, написанные для IBM-1401, гораздо быстрее, чем сама IBM-1401, поэтому не было никакого смысла в переделывании программ.

Компьютеры серии IBM-360 могли имитировать работу других компьютеров, потому что они создавались с использованием микропрограммирования. Нужно было всего лишь написать три микропрограммы: одну — для системы команд IBM-360, другую — для системы команд IBM-1401 и третью — для системы команд IBM-7094. Требование совместимости было одной из главных причин использования микропрограммирования.

IMB-360 удалось разрешить дилемму между двоичной и десятичной системой: у этого компьютера было 16 регистров по 32 бита для бинарной арифметики, но память состояла из байтов, как у IBM-1401. В ней использовались такие же команды для перемещения записей разного размера из одной части памяти в другую, как и в IBM-1401.

Объем памяти у IBM-360 составлял 2^{24} байтов (16 Мбайт). В те времена такой объем памяти казался огромным. Серия IBM-360 позднее сменилась серией IBM-370, затем IBM-4300, IBM-3080, IBM-3090. У всех этих компьютеров была сходная архитектура. К середине 80-х годов 16 Мбайт памяти стало недостаточно, и компании IBM пришлось частично отказаться от совместимости, чтобы перейти на систему адресов в 32 бита, необходимую для памяти объемом в 2^{32} байтов.

Можно было бы предположить, что поскольку у машин были слова в 32 бита и регистры, у них вполне могли бы быть и адреса в 32 бита. Но в то время никто не мог даже представить себе компьютер с объемом памяти 16 Мбайт. Обвинять IBM в отсутствии предвидения — все равно что обвинять современных производителей персональных компьютеров в том, что адреса в них всего по 32 бита. Возможно, через несколько лет объем памяти компьютеров будет составлять намного больше 4 Гбайт, и тогда адресов в 32 бита будет недостаточно.

Мир мини-компьютеров сделал большой шаг вперед в третьем поколении вместе с производством серии компьютеров PDP-11, последователей PDP-8 со

словами по 16 битов. Во многих отношениях PDP-11 был младшим братом IBM-360, а PDP-1 — младшим братом IBM-7094. И у IBM-360, и у PDP-11 были регистры, слова, память с байтами, и в обеих сериях были компьютеры разной стоимости и с разными функциями. PDP-1 широко использовался, особенно в университетах, и компания DEC продолжала лидировать среди производителей миникомпьютеров.

Четвертое поколение — сверхбольшие интегральные схемы (1980–?)

Появление **сверхбольших интегральных схем (СБИС)** в 80-х годах позволило помещать на одну плату сначала десятки тысяч, затем сотни тысяч и, наконец, миллионы транзисторов. Это привело к созданию компьютеров меньшего размера и с более высокой скоростью работы. До появления PDP-1 компьютеры были настолько большие и дорогостоящие, что компаниям и университетам приходилось иметь специальные отделы (**вычислительные центры**). К 80-м годам цены упали так сильно, что возможность приобретать компьютеры появилась не только у организаций, но и у отдельных людей. Началась эра персональных компьютеров.

Персональные компьютеры использовались совсем для других целей. Они применялись для обработки слов, а также для различных диалоговых прикладных программ, с которыми большие компьютеры не могли работать.

Первые персональные компьютеры продавались в виде комплектов. Каждый комплект содержал печатную плату, набор интегральных схем, обычно включающий схему Intel 8080, несколько кабелей, источник питания и иногда 8-дюймовый дисковод. Сложить из этих частей компьютер покупатель должен был сам. Программное обеспечение к компьютеру не прилагалось. Покупателю приходилось самому писать программное обеспечение. Позднее появилась операционная система CP/M, написанная Гари Килдаллом для Intel 8080. Эта действующая операционная система помещалась на дискету, она включала в себя систему управления файлами и интерпретатор для выполнения пользовательских команд, которые набирались с клавиатуры.

Еще один персональный компьютер, Apple (а позднее и Apple II), был разработан Стивом Джобсом и Стивом Возняком. Он стал чрезвычайно популярен среди отдельных покупателей, а также широко использовался в школах, и это сделало компанию Apple серьезным конкурентом IBM.

Наблюдая за тем, чем занимаются другие компании, компания IBM, лидирующая тогда на компьютерном рынке, тоже решила заняться производством персональных компьютеров. Но вместо того чтобы конструировать компьютер с нуля, что заняло бы слишком много времени, компания IBM предоставила одному из своих работников, Филипу Эстриджу, большую сумму денег, приказала ему отправиться куда-нибудь подальше от вмешивающихся во все бюрократов главного управления компании, находящегося в Нью-Йорке, и не возвращаться, пока не будет сконструирован действующий персональный компьютер. Эстридж открыл предприятие достаточно далеко от главного управления компании (во Флориде), взял Intel 8088 в качестве центрального процессора и создал персональный компьютер из серийных компонентов. Этот компьютер (IBM PC) появился в 1981 году и стал самым покупаемым компьютером в истории.

Но компания IBM сделала одну вещь, о которой она позже пожалела. Вместо того чтобы держать проект машины в секрете (или по крайней мере оградить себя патентами), как она обычно делала, компания опубликовала полные проекты, включая все электронные схемы, в книге стоимостью $49. Эта книга была опубликована для того, чтобы другие компании могли производить сменные платы для IBM PC, что повысило бы совместимость и популярность этого компьютера. К несчастью для IBM, как только проект IBM PC стал широко известен, поскольку все составные части компьютера можно было легко приобрести, многие компании начали делать **клоны** PC и часто продавали их гораздо дешевле, чем IBM. Так началось бурное производство персональных компьютеров.

Хотя некоторые компании (такие как Commodore, Apple, Amiga, Atari) производили персональные компьютеры с использованием своих процессоров, а не Intel, потенциал производства IBM PC был настолько велик, что другим компаниям приходилось пробиваться с трудом. Выжить удалось только некоторым из них, и то потому, что они специализировались в узких областях, например в производстве рабочих станций или суперкомпьютеров.

Первая версия IBM PC была оснащена операционной системой MS-DOS, которую выпускала тогда еще крошечная корпорация Microsoft. IBM и Microsoft совместно разработали последовавшую за MS-DOS операционную систему OS/2, характерной чертой которой был графический интерфейс, сходный с интерфейсом Apple Macintosh. Между тем компания Microsoft также разработала собственную операционную систему Windows, которая работала на основе MS-DOS, на случай, если OS/2 не будет иметь спроса. OS/2 действительно не пользовалась спросом, а Microsoft успешно продолжала выпускать операционную систему Windows, что послужило причиной грандиозного раздора между IBM и Microsoft. Легенда о том, как крошечная компания Intel и компания Microsoft, которая была еще меньше, чем Intel, умудрились свергнуть IBM, одну из самых крупных, самых богатых и самых влиятельных корпораций в мировой истории, подробно излагается в бизнес-школах по всему миру.

В середине 80-х годов на смену CISC[1] пришел RISC[2]. Команды RISC были проще и работали гораздо быстрее. В 90-х годах появились суперскалярные процессоры, которые могли выполнять много команд одновременно, часто не в том порядке, в котором они появляются в программе. Мы введем понятия RISC, CISC и суперскалярного процессора в главе 2 и обсудим их подробно.

Типы компьютеров

В предыдущем разделе мы кратко изложили историю компьютерных систем. В этом разделе мы расскажем о положении дел в настоящий момент и сделаем некоторые предположения на будущее. Хотя наиболее известны персональные компьютеры, в наши дни существуют и другие типы машин, поэтому стоит кратко рассказать о них.

[1] Complex instruction set computer — компьютер на микропроцессоре с полным набором команд. — *Примеч. перев.*

[2] Reduced instruction set computer — компьютер с сокращенным набором команд. — *Примеч. перев.*

Технологические и экономические аспекты

Компьютерная промышленность двигается вперед как никакая другая. Главная движущая сила — способность производителей помещать с каждым годом все больше и больше транзисторов на микросхему. Чем больше транзисторов (крошечных электронных переключателей), тем больше объем памяти и мощнее процессоры.

Степень технологического прогресса можно наблюдать, используя **закон Мура**, названный в честь одного из основателей и главы компании Intel Гордона Мура, который открыл его в 1965 году. Когда Мур готовил доклад для промышленной группы, то заметил, что каждое новое поколение микросхем появляется через три года после предыдущего. Поскольку у каждого нового поколения компьютеров было в 4 раза больше памяти, чем у предыдущего, он понял, что число транзисторов на микросхеме возрастает на постоянную величину, и, таким образом, этот рост можно предсказывать на годы вперед. Закон Мура гласит, что число транзисторов на одной микросхеме удваивается каждые 18 месяцев, то есть увеличивается на 60% каждый год. Размеры микросхем и даты их производства, показанные на рис. 1.6, подтверждают, что закон Мура до сих пор действует.

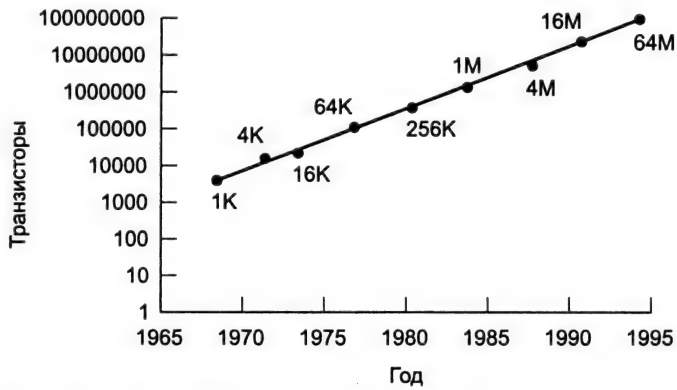


Рис. 1.6. Закон Мура предсказывает, что количество транзисторов на одной микросхеме увеличивается на 60% каждый год. Точки на графике — размер памяти в битах

Конечно, закон Мура — это вообще не закон, а просто эмпирическое наблюдение о том, с какой скоростью физики и инженеры-технологи развивают компьютерные технологии, и предсказание, что с такой скоростью они будут работать и в будущем. Многие специалисты считают, что закон Мура действует и в XXI веке, возможно, до 2020 года. Вероятно, транзисторы скоро будут состоять всего лишь из нескольких атомов, хотя достижения квантовой компьютерной техники, может быть, позволят использовать для размещения 1 бита спин одного электрона.

Закон Мура связан с тем, что некоторые экономисты называют **эффективным циклом**. Достижения в компьютерных технологиях (увеличение количества транзисторов на одной микросхеме) приводят к продукции лучшего качества и более низким ценам. Низкие цены ведут к появлению новых прикладных программ (никому не приходило в голову разрабатывать компьютерные игры, когда каждый компьютер стоил $10 млн). Новые прикладные программы приводят к возникновению новых компьютерных рынков и новых компаний. Существование всех этих

компаний ведет к конкуренции между ними, которая, в свою очередь, создает экономический спрос на лучшие технологии. Круг замыкается.

Еще один фактор развития компьютерных технологий — первый натановский закон программного обеспечения, названный в честь Натана Мирвольда, главного администратора компании Microsoft. Этот закон гласит: «Программное обеспечение — это газ. Оно распространяется и полностью заполняет резервуар, в котором находится». В 80-е годы электронная обработка текстов осуществлялась программой troff (программа troff использовалась при создании этой книги). Troff занимает несколько десятков килобайтов памяти. Современные электронные редакторы занимают десятки мегабайтов. В будущем, несомненно, они будут занимать десятки гигабайтов. Программное обеспечение продолжает развиваться и создает постоянный спрос на процессоры, работающие с более высокой скоростью, на больший объем памяти, на большую производительность устройств ввода-вывода.

С каждым годом происходит стремительное увеличение количества транзисторов на одной микросхеме. Отметим, что достижения в развитии других частей компьютера столь же велики. Например, у IBM PC/XT, появившегося в 1982 году, объем жесткого диска составлял всего 10 Мбайт, гораздо меньше, чем у большинства современных настольных компьютеров. Подсчитать, насколько быстро происходит совершенствование жесткого диска, гораздо сложнее, поскольку тут есть несколько параметров (объем, скорость передачи данных, цена и т. д.), но измерение любого из этих параметров покажет, что показатели возрастают, по крайней мере, на 50% в год.

Крупные достижения наблюдаются также и в сфере телекоммуникаций и создания сетей. Меньше чем за два десятилетия мы пришли от модемов, передающих информацию со скоростью 300 бит/с, к аналоговым модемам, работающим со скоростью 56 Кбит/с, телефонным линиям ISDN, где скорость передачи информации 2×64 Кбит/с, оптико-волоконным сетям, где скорость уже больше чем 1 Гбит/с. Оптико-волоконные трансатлантические телефонные кабели (например, TAT-12/13) стоят около $700 млн, действуют в течение 10 лет и могут передавать 300 000 звонков одновременно, поэтому стоимость 10-минутной межконтинентальной связи составляет менее 1 цента. Лабораторные исследования подтвердили, что возможны системы связи, работающие со скоростью 1 терабит/с (10^{12} бит/с) на расстоянии более 100 км без усилителей. Едва ли нужно упоминать здесь о развитии сети Интернет.

Широкий спектр компьютеров

Ричард Хамминг, бывший исследователь из Bell Laboratories, заметил, что количественное изменение величины на порядок ведет к качественному изменению. Например, гоночная машина, которая может ездить со скоростью 1000 км/ч по пустыне Невада, коренным образом отличается от обычной машины, которая ездит со скоростью 100 км/ч по шоссе. Точно так же небоскреб в 100 этажей несопоставим с десятиэтажным многоквартирным домом. А если речь идет о компьютерах, то тут за три десятилетия количественные показатели увеличились не в 10, а в 1 000 000 раз.

Развивать компьютерные технологии можно двумя путями: или создавать компьютеры все большей и большей мощности при постоянной цене, или выпускать один и тот же компьютер, с каждым годом снижая цену. Компьютерная промышленность использует оба эти пути, создавая широкий спектр разнообразных компьютеров. Очень приблизительная классификация современных компьютеров представлена в табл. 1.3.

Таблица 1.3. Типы современных компьютеров. Указанные цены приблизительны

Тип	Цена ($)	Сфера применения
«Одноразовые» компьютеры	1	Поздравительные открытки
Встроенные компьютеры	10	Часы, машины, различные приборы
Игровые компьютеры	100	Домашние компьютерные игры
Персональные компьютеры	1000	Настольные и портативные компьютеры
Серверы	10 000	Сетевые серверы
Рабочие станции	100 000	Мини-суперкомпьютеры
Большие компьютеры	1 000 000	Обработка пакетных данных в банке
Суперкомпьютеры	10 000 000	Предсказание погоды на длительный срок

В самой верхней строчке находятся микросхемы, которые приклеиваются на внутреннюю сторону поздравительных открыток для проигрывания мелодий «Happy Birthday», свадебного марша или чего-нибудь подобного. Автор идеи еще не придумал открытки с соболезнованиями, которые играют похоронный марш, но поскольку он выпустил эту идею в потребительскую сферу, вскоре можно будет ожидать появления и таких открыток. Тот, кто воспитывался на компьютерах стоимостью в миллионы долларов, воспринимает такие доступные всем компьютеры примерно так же, как доступный всем самолет. Тем не менее такие компьютеры, вне всяких сомнений, должны существовать (а как насчет говорящих мешков для мусора, которые просят вас не выбрасывать алюминиевые банки?).

Вторая строчка — компьютеры, которые помещаются внутрь телефонов, телевизоров, микроволновых печей, CD-плейеров, игрушек, кукол и т. п. Через несколько лет во всех электрических приборах будут находиться встроенные компьютеры, количество которых будет измеряться в миллиардах. Такие компьютеры состоят из процессора, памяти менее 1 Мбайт и устройств ввода-вывода, и все это на одной маленькой микросхеме, которая стоит всего несколько долларов.

Следующая строка — игровые компьютеры. Это обычные компьютеры с особой графикой, но с ограниченным программным обеспечением и почти полным отсутствием открытости, то есть возможности перепрограммирования. Примерно равны им по стоимости электронные записные книжки и прочие карманные компьютеры, а также сетевые компьютеры и web-терминалы. Все они содержат процессор, несколько мегабайтов памяти, какой-либо дисплей (может быть, даже телевизионный) и больше ничего. Поэтому они такие дешевые.

Далее идут персональные компьютеры. Именно они ассоциируются у большинства людей со словом «компьютер». Персональные компьютеры бывают двух видов: настольные и ноутбуки. Они обычно содержат несколько мегабайтов памяти, жесткий диск с данными на несколько гигабайтов, CD-ROM, модем, звуковую карту

и другие периферийные устройства. Они снабжены сложными операционными системами, имеют возможность наращивания, при работе с ними используется широкий спектр программного обеспечения. Компьютеры с процессором Intel обычно называются «персональными компьютерами», а компьютеры с другими процессорами — «рабочими станциями», хотя особой разницы между ними нет.

Персональные компьютеры и рабочие станции часто используются в качестве сетевых серверов как для локальных сетей (обычно в пределах одной организации), так и для Интернета. У этих компьютеров обычно один или несколько процессоров, несколько гигабайтов памяти и много Гбайт на диске. Такие компьютеры способны работать в сети с очень высокой скоростью. Некоторые из них могут обрабатывать тысячи поступающих сообщений одновременно.

Помимо небольших серверов с несколькими процессорами существуют системы, которые называются **сетями рабочих станций (NOW — Networks of Workstations)** или **кластерами рабочих станций (COW — Clusters of Workstations)**. Они состоят из обычных персональных компьютеров или рабочих станций, связанных в сеть, по которой информация передается со скоростью 1 Гбит/с, и специального программного обеспечения, позволяющего всем машинам одновременно работать над одной задачей. Такие системы широко применяются в науке и технике. Кластеры рабочих станций могут включать в себя от нескольких компьютеров до нескольких тысяч. Благодаря низкой цене компонентов отдельные организации могут приобретать такие машины, которые по эффективности являются мини-суперкомпьютерами.

А теперь мы дошли до больших компьютеров размером с комнату, напоминающих компьютеры 60-х годов. В большинстве случаев эти системы — прямые потомки больших компьютеров серии IBM-360. Обычно они работают ненамного быстрее, чем мощные серверы, но у них выше скорость процессов ввода-вывода и обладают они довольно большим пространством на диске — 1 терабайт и более (1 терабайт=10^{12} байт). Такие системы стоят очень дорого и требуют крупных вложений в программное обеспечение, данные и персонал, обслуживающий эти компьютеры. Многие компании считают, что дешевле заплатить несколько миллионов долларов один раз за такую систему, чем даже думать о том, что нужно будет заново программировать все прикладные программы для маленьких компьютеров.

Именно этот класс компьютеров привел к проблеме 2000 года. Проблема возникла из-за того, что в 60-е и 70-е годы программисты, пишущие программы на языке COBOL, представляли год двузначным десятичным числом с целью экономии памяти. Они не смогли предвидеть, что их программное обеспечение будет использоваться через три или четыре десятилетия. Многие компании повторили ту же ошибку, добавив к числу года только два десятичных разряда. Автор этой книги предсказывает, что конец цивилизации произойдет в полночь 31 декабря 9999 года, когда сразу уничтожатся все COBOL-программы, написанные за 8000 лет[1].

Вслед за большими компьютерами идут настоящие суперкомпьютеры. Их процессоры работают с очень высокой скоростью, объем памяти у них составляет множество гигабайтов, диски и сети также работают очень быстро. В последние годы многие суперкомпьютеры стали очень похожи, они почти не отличаются от кластеров рабочих станций, но у них больше составляющих и они работают быстрее.

[1] Необходимо отметить, что в полночь 31 декабря 1999 года катастрофы не произошло. — *Примеч. перев.*

Суперкомпьютеры используются для решения различных научных и технических задач, которые требуют сложных вычислений, например таких, как моделирование сталкивающихся галактик, разработка новых лекарств, моделирование потока воздуха вокруг крыла аэроплана.

Семейства компьютеров

В этом разделе мы дадим краткое описание трех компьютеров, которые будут использоваться в качестве примеров в этой книге: Pentium II, UltraSPARC II и picoJava II.

Pentium II

В 1968 году Роберт Нойс, изобретатель кремниевой интегральной схемы, Гордон Мур, автор известного закона Мура, и Артур Рок, капиталист из Сан-Франциско, основали корпорацию Intel для производства компьютерных микросхем. За первый год своего существования корпорация продала микросхем всего на $3000, но потом объем продаж компании заметно увеличился.

В конце 60-х годов калькуляторы представляли собой большие электромеханические машины размером с современный лазерный принтер и весили около 20 кг. В сентябре 1969 года японская компания Busicom обратилась к корпорации Intel с просьбой выпустить 12 несерийных микросхем для электронной вычислительной машины. Инженер компании Intel Тед Хофф, назначенный на выполнение этого проекта, решил, что можно поместить 4-битный универсальный процессор на одну микросхему, которая будет выполнять те же функции и при этом окажется проще и дешевле. Так в 1970 году появился первый процессор на одной микросхеме, процессор 4004 на 2300 транзисторах.

Заметим, что ни Intel, ни Busicom не имели ни малейшего понятия, какое грандиозное открытие они совершили. Когда компания Intel решила, что стоит попробовать использовать процессор 4004 в других разработках, она предложила купить все права на новую микросхему у компании Busicom за $60000, то есть за сумму, которую Busicom заплатила Intel за разработку этой микросхемы. Busicom сразу приняла предложение Intel, и Intel начала работу над 8-битной версией микросхемы 8008, выпущенной в 1972 году.

Компания Intel не ожидала большого спроса на микросхему 8008, поэтому она выпустила небольшое количество этой продукции. Ко всеобщему удивлению, новая микросхема вызвала большой интерес, поэтому Intel начала разработку еще одного процессора, в котором предел в 16 Кбайт памяти (как у процессора 8008), навязываемый количеством внешних выводов микросхемы, был преодолен. Так появился небольшой универсальный процессор 8080, выпущенный в 1974 году. Как и PDP-8, он произвел революцию на компьютерном рынке и сразу стал массовым продуктом: только компания DEC продала тысячи PDP-8, а Intel — миллионы процессоров 8080.

В 1978 году появился процессор 8086 — 16-битный процессор на одной микросхеме. Процессор 8086 был во многом похож на 8080, но не был полностью совместим с ним. Затем появился процессор 8088 с такой же архитектурой, как и у 8086.

Он выполнял те же программы, что и 8086, но вместо 16-битной шины у него была 8-битная, из-за чего процессор работал медленнее, но стоил дешевле, чем 8086[1]. Когда IBM выбрала процессор 8088 для IBM PC, эта микросхема стала эталоном в производстве персональных компьютеров.

Ни 8088, ни 8086 не могли обращаться к более 1 Мбайт памяти. К началу 80-х годов это стало серьезной проблемой, поэтому компания Intel разработала модель 80286, совместимую с 8086. Основной набор команд остался в сущности таким же, как у процессоров 8086 и 8088, но память была устроена немного по-другому, хотя и могла работать по-прежнему из-за требования совместимости с предыдущими микросхемами. Процессор 80286 использовался в IBM PC/AT и в моделях PS/2. Он, как и 8088, пользовался большим спросом (главным образом потому, что покупатели рассматривали его как более быстрый процессор 8088).

Следующим шагом был 32-битный процессор 80386, выпущенный в 1985 году. Как и 80286, он был более или менее совместим со всеми старыми версиями. Совместимость такого рода оказывалась благом для тех, кто пользовался старым программным обеспечением, и некоторым неудобством для тех, кто предпочитал современную архитектуру, не обремененную ошибками и технологиями прошлого.

Через четыре года появился процессор 80486. Он работал быстрее, чем 80386, мог выполнять операции с плавающей точкой и имел 8 Кбайт кэш-памяти. **Кэш-память** используется для того, чтобы держать наиболее часто используемые слова внутри центрального процессора и избегать длительного доступа к основной (оперативной) памяти. Иногда кэш-память находится не внутри центрального процессора, а рядом с ним. 80486 содержал встроенные средства поддержки многопроцессорного режима, что давало производителям возможность конструировать системы с несколькими процессорами.

В этот момент Intel, проиграв судебную тяжбу по поводу нарушения правил наименования товаров, выяснила, что номера (например, 80486) не могут быть торговой маркой, поэтому следующее поколение компьютеров получило название Pentium (от греческого слова πεντε — пять). В отличие от 80486, у которого был один внутренний конвейер, Pentium имел два, что позволяло работать ему почти в два раза быстрее (конвейеры мы рассмотрим подробно в главе 2).

Когда появилось следующее поколение компьютеров, те, кто рассчитывал на название Sexium (sex по-латыни — шесть), были разочарованы. Название Pentium стало так хорошо известно, что его решили оставить, и новую микросхему назвали Pentium Pro. Несмотря на столь незначительное изменение названия, этот процессор очень сильно отличался от предыдущего. У него была совершенно другая внутренняя организация, и он мог выполнять до пяти команд одновременно.

Еще одно нововведение у Pentium Pro — двухуровневая кэш-память. Процессор содержал 8 Кбайт памяти для часто используемых команд и еще 8 Кбайт для часто используемых данных. В корпусе Pentium Pro рядом с процессором (но не на самой микросхеме) находилась другая кэш-память в 256 Кбайт.

---

[1] На самом деле разница в стоимости самих микропроцессоров была незначительной. Но компьютеры, собираемые на базе микропроцессора 8088, были дешевле, чем если бы их строили на базе микропроцессора 8086. В то время были распространены 8-битные периферийные устройства, поэтому микропроцессор 8088 позволял упростить сопряжение с внешними устройствами. — *Примеч. научн. ред.*

Вслед за Pentium Pro появился процессор Pentium II, по существу такой же, как и его предшественник, но с особой системой команд для мультимедиа-задач (**MMX** — multimedia extensions). Эта система команд предназначалась для ускорения вычислений, необходимых при воспроизведении изображения и звука. При наличии MMX специальные сопроцессоры были не нужны. Данные команды имелись в наличии и в более поздних версиях Pentium, но их не было в Pentium Pro. Таким образом, компьютер Pentium II сочетал в себе функции Pentium Pro с мультимедиа-командами.

В начале 1998 года Intel запустил новую линию продукции под названием Celeron. Celeron имел меньшую производительность, чем Pentium II, но зато стоил дешевле. Поскольку у компьютера Celeron такая же архитектура, как у Pentium II, мы не будем обсуждать его в этой книге. В июне 1998 года компания Intel выпустила специальную версию Pentium II — Xeon. Он имел кэш-память большего объема, его внутренняя шина работала быстрее, были усовершенствованы средства поддержки многопроцессорного режима, но во всем остальном он остался обычным Pentium II, поэтому мы его тоже не будем обсуждать. Компьютеры семейства Intel показаны в табл. 1.4.

Таблица 1.4. Семейство процессоров Intel. Тактовая частота измеряется в МГц (1 МГц = 1 млн циклов/с)

Микросхема	Дата выпуска	Тактовая частота, МГц	Количество транзисторов	Объем памяти	Примечания
4004	4/1971	0,108	2 300	640 Кбайт	Первый микропроцессор на микросхеме
8008	4/1972	0,08	3 500	16 Кбайт	Первый 8-битный микропроцессор
8080	4/1974	2	6 000	64 Кбайт	Первый многоцелевой процессор на микросхеме
8086	6/1978	5–10	29 000	1 Мбайт	Первый 16-битный процессор на микросхеме
8088	6/1979	5–8	29 000	1 Мбайт	Использовался в IBM PC
80286	2/1982	8–12	134 000	1 Мбайт	Появилась защита памяти
80386	10/1985	16–33	275 000	4 Гбайт	Первый 32-битный процессор
80486	4/1989	25–100	1 200 000	4 Гбайт	8 Кбайт кэш-памяти
Pentium	3/1993	60–223	3 100 000	4 Гбайт	Два конвейера, у более поздних моделей — MMX
Pentium Pro	3/1995	150–200	5 500 000		Два уровня кэш-памяти
Pentium II	5/1997	233–400	7 500 000	64 Гбайт	Pentium Pro + MMX

[1] Шина адреса у микропроцессоров Pentium Pro и Pentium II имеет ширину 36 битов, что позволяет адресовать непосредственно 64 Гбайт. — *Примеч. научн. ред.*

Все микросхемы Intel совместимы со своими предшественниками вплоть до процессора 8086. Другими словами, Pentium II может выполнять программы, написанные для процессора 8086[1]. Совместимость всегда была одним из главных требований при разработке новых компьютеров, чтобы покупатели могли продолжать работать со старым программным обеспечением и не тратить деньги на новое. Конечно, Pentium II во много раз сложнее, чем 8086, поэтому он может выполнять многие функции, которые не способен выполнять процессор 8086. Все эти постепенные доработки в каждой новой версии привели к тому, что архитектура Pentium II не так проста, как могла бы быть, если бы разработчикам процессора Pentium II предоставили 7,5 млн транзисторов и команд, чтобы начать все заново.

Интересно, что хотя закон Мура раньше ассоциировался с числом битов в памяти компьютера, он в равной степени применим и по отношению к процессорам. Если напротив даты выпуска каждой микросхемы поставить число транзисторов на этой микросхеме (количество транзисторов показано в табл. 1.4), мы увидим, что закон Мура действует и здесь. График показан на рис. 1.7.

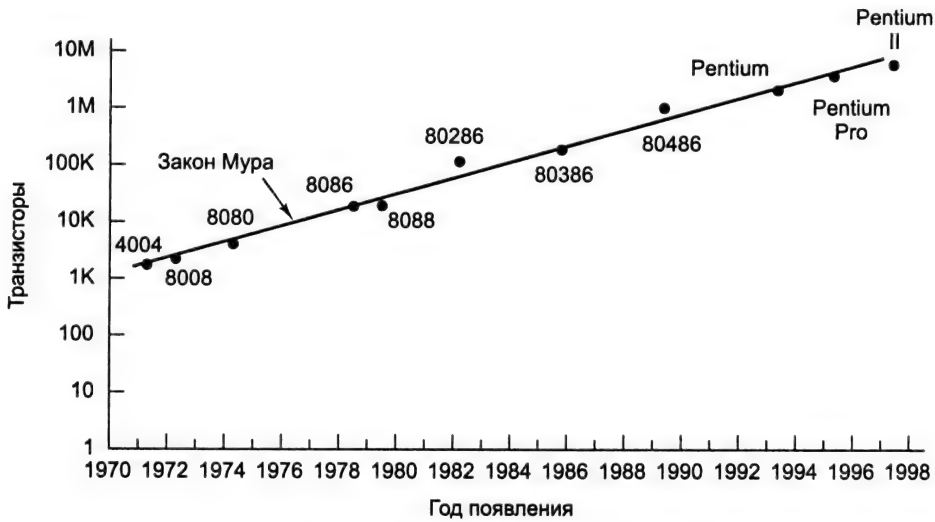


Рис. 1.7. Закон Мура действует и для процессоров

UltraSPARC II

В 70-х годах во многих университетах была очень популярна операционная система UNIX, но персональные компьютеры не подходили для этой операционной системы, поэтому любителям UNIX приходилось работать на мини-компьютерах с разделением времени, таких как PDP-11 и VAX. Энди Бехтольсхайм, аспирант Стэнфордского университета, был очень расстроен тем, что ему нужно посещать

[1] Существуют сотни и тысячи программ, которые не могут быть выполнены на современных быстродействующих микропроцессорах, совместимых с микропроцессором 8086, хотя на более старых (медленных) микропроцессорах они и выполняются. — *Примеч. научн. ред.*

компьютерный центр, чтобы работать с UNIX. В 1981 году он разрешил эту проблему, самостоятельно построив персональную рабочую станцию UNIX из стандартных частей, имеющихся в продаже, и назвал ее SUN-1 (Stanford University Network — сеть Стэнфордского университета).

На Бехтольсхайма скоро обратил внимание Винод Косла, 27-летний индиец, который горел желанием годам к тридцати стать миллионером и уйти от дел. Косла предложил Бехтольсхайму организовать компанию по производству рабочих станций Sun. Он нанял Скота Мак-Нили, другого аспиранта Стэнфордского университета, чтобы тот возглавил производство. Для написания программного обеспечения они наняли Билла Джоя, главного создателя системы UNIX. В 1982 году они вчетвером основали компанию Sun Microsystems. Первый компьютер компании, Sun-1, был оснащен процессором Motorola 68020 и имел большой успех, как и последующие модели Sun-2 и Sun-3, которые также были сконструированы с использованием микропроцессоров Motorola. Эти машины были гораздо мощнее, чем другие персональные компьютеры того времени (отсюда и название «рабочая станция»), и изначально были предназначены для работы в сети. Каждая рабочая станция Sun была оснащена сетевым адаптером Ethernet и программным обеспечением TCP/IP для связи с сетью ARPANET, предшественницей Интернета.

В 1987 году компания Sun, которая к тому времени продавала рабочих станций на полмиллиарда долларов в год, решила разработать свой собственный процессор, основанный на новом революционном проекте калифорнийского университета в Беркли (RISC II). Этот процессор назывался **SPARC (Scalable Processor ARCitecture — наращиваемая архитектура процессора)**. Он был использован при производстве рабочей станции Sun-4. Через некоторое время все рабочие станции компании Sun стали производиться на основе этого процессора.

В отличие от многих других компьютерных компаний, Sun решила не заниматься производством процессоров SPARC. Вместо этого она предоставила патент на их изготовление нескольким предприятиям, надеясь, что конкуренция между ними повлечет за собой повышение качества продукции и снижение цен. Эти предприятия выпустили несколько разных микросхем, основанных на разных технологиях, работающих с разной скоростью и отличающихся друг от друга по стоимости. Микросхемы назывались MicroSPARC, HyperSPARK, SuperSPARK и TurboSPARK. Мало чем отличаясь друг от друга, все они были совместимы и могли выполнять одни и те же программы, которые не приходилось изменять.

Компания Sun всегда хотела, чтобы разные предприятия поставляли для SPARK составные части и системы. Нужно было построить целую индустрию, только в этом случае можно было конкурировать с компанией Intel, лидирующей на рынке персональных компьютеров. Чтобы завоевать доверие компаний, которые были заинтересованы в производстве процессоров SPARC, но не хотели вкладывать средства в продукцию, которую будет подавлять Intel, компания Sun создала промышленный консорциум SPARC International для руководства развитием будущих версий архитектуры SPARC. Важно различать архитектуру SPARC, которая представляет собой набор команд, и собственно выполнение этих команд. В этой книге мы будем говорить и об общей архитектуре SPARC, и о процессоре, используемом в рабочей станции SPARC (предварительно обсудив процессоры в третьей и четвертой главах).

Первый SPARC был 32-битным и работал с частотой 36 МГц. Центральный процессор назывался **IU (Integer Unit — процессор целочисленной арифметики)** и был весьма посредственным. У него было только три основных формата команд и в общей сложности всего 55 команд. С появлением процессора с плавающей точкой добавилось еще 14 команд. Отметим, что компания Intel начала с 8- и 16-битных микросхем (модели 8088, 8086, 80286), а уже потом перешла на 32-битные (модель 80386), а Sun, в отличие от Intel, сразу начала с 32-битных.

Грандиозный перелом в развитии SPARC произошел в 1995 году, когда была разработана 64-битная версия (версия 9) с адресами и регистрами по 64 бит. Первой рабочей станцией с такой архитектурой стал UltraSPARC I, вышедший в свет в 1995 году. Он был полностью совместим с 32-битными версиями SPARC, хотя сам был 64-битным.

В то время как предыдущие машины работали с символьными и числовыми данными, UltraSPARC с самого начала был предназначен для работы с изображениями, аудио, видео и мультимедиа вообще. Среди нововведений, помимо 64-битной архитектуры, появились 23 новые команды, в том числе команды для упаковки и распаковки пикселов из 64-битных слов, масштабирования и вращения изображений, перемещения блоков, а также для компрессии и декомпрессии видео в реальном времени. Эти команды назывались **VIS (Visual Instruction Set)** и предназначались для поддержки мультимедиа. Они были аналогичны командам MMX.

UltraSPARC предназначался для web-серверов с десятками процессоров и физической памятью до 2 Тбайт (терабайт, 1Тбайт = 10^{12} байтов). Тем не менее некоторые версии UltraSPARC могут использоваться и в ноутбуках.

За UltraSPARC I последовали UltraSPARC II и UltraSPARC III. Эти модели отличались друг от друга по скорости, и у каждой из них появлялись какие-то новые особенности. Когда мы будем говорить об архитектуре SPARC, мы будем иметь в виду 64-битную версию компьютера UltraSPARC II (версии 9).

PicoJava II

Язык программирования C придумал один из работников компании Bell Laboratories Деннис Ритчи. Этот язык предназначался для работы в операционной системе UNIX. Из-за большой популярности UNIX C скоро стал доминирующим языком в системном программировании. Через несколько лет Бьярн Строуструп, тоже из компании Bell Laboratories, добавил к C некоторые особенности из объектно-ориентированного программирования, и появился язык C++, который также стал очень популярным.

В середине 90-х годов исследователи в Sun Microsystems думали, как сделать так, чтобы пользователи могли вызывать двоичные программы через Интернет и загружать их как часть web-страниц. Им нравился C++, но он не был надежным в том смысле, что программа, посланная на некоторый компьютер, могла причинить ущерб этому компьютеру. Тогда они решили на основе C++ создать новый язык программирования Java, с которым не было бы подобных проблем. Java — объектно-ориентированный язык, который применяется при решении различных прикладных задач. Поскольку этот язык прост и популярен, мы будем использовать его для примеров.

Поскольку Java — всего лишь язык программирования, можно написать компилятор, который будет преобразовывать его для Pentium, SPARC или любого другого компьютера. Такие компиляторы существуют. Однако этот язык был создан в первую очередь для того, чтобы пересылать программы между компьютерами по Интернету и чтобы пользователям не приходилось изменять их. Но если программа на языке Java компилировалась для SPARC, то когда она пересылалась по Интернету на Pentium, запустить там эту программу было уже нельзя.

Чтобы разрешить эту проблему, компания Sun придумала новую виртуальную машину **JVM (Java Virtual Machine — виртуальная машина Java)**. Память у этой машины состояла из 32-битных слов, машина поддерживала 226 команд. Большинство команд были простыми, но выполнение некоторых довольно сложных команд требовало большого количества циклов обращения к памяти.

В компании Sun разработали компилятор, преобразующий программы на языке Java на уровень JVM, и интерпретатор JVM для выполнения этих программ. Этот интерпретатор был написан на языке C и, значит, мог использоваться практически на любом компьютере. Следовательно, чтобы компьютер мог выполнять двоичные программы на языке Java, нужно было всего лишь достать интерпретатор JVM для соответствующего компьютера (например, для Pentium II с системой Windows 98 или для SPARC с системой UNIX) вместе с определенными программами поддержки и библиотеками. Кроме того, большинство браузеров в Интернете содержат интерпретатор JVM, что позволяет легко запускать апплеты (небольшие двоичные программы на Java, связанные со страницами World Wide Web). Большинство этих апплетов поддерживают анимацию и звук.

Интерпретация программ JVM (и любых других программ) происходит медленно. Альтернативный подход — сначала скомпилировать апплет или другую программу JVM для вашей собственной машины, а затем запустить скомпилированную программу. Такая стратегия требует наличия компилятора с JVM на машинный язык внутри браузера и возможности активизировать его, когда необходимо. Эти компиляторы называются **JIT-компиляторами (Just In Time** — «как раз вовремя»), и они широко распространены. Однако эта система создает некоторую задержку между получением JVM-программы и ее выполнением, поскольку JVM-программа компилируется на машинный язык.

Кроме программного обеспечения JVM (JVM-интерпретаторов и JIT-компиляторов) Sun и другие компании разработали микросхемы JVM — процессоры, которые сразу выполняют двоичные программы JVM без какой-либо интерпретации и компиляции. PicoJava I и picoJava II были разработаны для рынка встроенных систем. На этом рынке требуются мощные и очень дешевые процессоры (цена ниже $50), встраиваемые внутрь пластиковых карточек, телевизоров, телефонов и других устройств, особенно таких, которые обеспечивают связь с внешним миром. Предприятия, имеющие патент на производство микросхем компании Sun, могли производить собственные микросхемы на основе проекта picoJava, в той или иной степени изменяя их, включая и убирая процессор с плавающей точкой, преобразуя размер кэш-памяти и т. п.

Ценность микросхемы Java состоит в том, что она способна менять функции в процессе работы. Например, представим себе администратора, у которого есть телефон с процессором Java. Администратору никогда не приходилось читать фак-

сы на крошечном экране телефона, но в один прекрасный день ему это понадобилось. Тогда он звонит провайдеру и просит предоставить ему апплет для просмотра факсов, и таким образом добавляет новую функцию к своему телефону. Но из-за некоторых особенностей прибора и недостатка памяти невозможно использовать интерпретаторы и JIT-компиляторы, поэтому именно в таких случаях необходимы микросхемы JVM.

PicoJava II — не физическая микросхема (вы не можете пойти в магазин и купить ее), а проект, который является основой для ряда микросхем, например Sun MicroJava 701 и других. Эти микросхемы производятся предприятиями, получившими патент Sun. Мы будем использовать процессор picoJava II в качестве иллюстративного примера, поскольку он очень сильно отличается от Pentium II и UltraSPARC II и имеет совершенно другую сферу применения. PicoJava II представляет особый интерес для нас, поскольку в главе 4 мы расскажем, как можно создать JVM с помощью микропрограммирования. Тогда мы сможем сравнить спрограммированный JVM с аппаратным обеспечением JVM.

PicoJava II содержит два факультативных процессора: кэш-память и процессор с плавающей точкой, которые каждый производитель может включать или не включать в разработку. В целях простоты мы будем рассматривать picoJava II как микросхему, хотя на самом деле это не микросхема, а проект микросхемы. Иногда мы будем говорить о микросхеме Sun MicroJava 701, которая является воплощением проекта picoJava II. Но даже если мы не будем упоминать конкретные микросхемы, читатели должны помнить, что picoJava II — это не физическая микросхема, а проект, на основе которого производители разрабатывают разные микросхемы.

Используя Pentium II, UltraSPARC II и picoJava II в качестве примеров, мы можем изучить три разных типа процессоров. Первый из них представляет собой CISC с суперскалярным процессором, второй — RISC с суперскалярным процессором. Третий используется во встроенных системах. Эти три процессора сильно отличаются друг от друга, что дает нам возможность лучше увидеть диапазон компьютерных разработок.

Краткое содержание книги

Эта книга о многоуровневых компьютерах и о том, как они организованы (отметим, что почти все современные компьютеры многоуровневые). Подробно мы рассмотрим четыре уровня — цифровой логический уровень, микроархитектурный уровень, уровень архитектуры набора команд и уровень операционной системы. Основные вопросы, которые будут обсуждаться в этой книге, включают общую структуру уровней (и почему уровни построены именно таким образом), типы команд и данных, организацию памяти, адресацию, а также способы построения каждого уровня. Все это называется компьютерной организацией или компьютерной архитектурой.

Мы в первую очередь имеем дело с общими понятиями и не касаемся деталей и строгой математики. По этой причине многие примеры будут сильно упрощены, чтобы сделать упор на основные понятия, а не на детали.

Чтобы разъяснить, как принципы, изложенные в этой книге, могут применяться на практике, мы в качестве примеров будем использовать компьютеры Pentium II, UltraSPARC II и picoJava II. Они были выбраны по нескольким причинам. Во-первых, они широко используются, и у читателя наверняка есть доступ хотя бы к одному из них. Во-вторых, каждый из этих компьютеров обладает собственной уникальной архитектурой, что дает основу для сравнения и возможность показать альтернативные варианты. Книги, в которых рассматривается только один компьютер, оставляют у читателя чувство, будто это и есть единственный нормальный компьютер, что является абсурдным в свете огромного числа компромиссов и произвольных решений, которые разработчики вынуждены принимать. Читатель должен рассматривать эти и все другие компьютеры критически и постараться понять, почему дела обстоят именно таким образом и что можно было бы изменить, а не просто принимать их как данность.

Нужно уяснить с самого начала, что эта книга не о том, как программировать Pentium II, UltraSPARC II и picoJava II. Эти компьютеры используются только в качестве иллюстративных примеров, и мы не претендуем на их полное описание. Читателям, желающим ознакомиться с этими компьютерами, следует обратиться к публикациям производителей.

Глава 2 знакомит читателей с основными компонентами компьютера: процессорами, памятью, устройствами ввода-вывода. В ней дается краткое описание системной архитектуры и введение к следующим главам.

Главы 3, 4, 5 и 6 касаются каждая одного из уровней, показанных на рис. 1.2. Мы идем снизу вверх, поскольку компьютеры разрабатывались именно таким образом. Структура уровня k в значительной степени определяется особенностями уровня k–1, поэтому очень трудно понять, как устроен определенный уровень, если не рассмотреть подробно предыдущий, который и определил строение последующего. К тому же с точки зрения обучения логичнее следовать от более простых уровней к более сложным, а не наоборот.

Глава 3 посвящена цифровому логическому уровню, то есть аппаратному обеспечению. В ней рассказывается, что такое вентили и как они объединяются в схемы. В этой главе также вводятся основные понятия булевой алгебры, которая используется для обработки цифровых данных. Кроме того, объясняется, что такое шины, причем особое внимание уделяется популярной шине PCI. В главе приводится много разнообразных примеров, в том числе относящихся к трем компьютерам, упомянутым выше.

Глава 4 знакомит читателя со строением микроархитектурного уровня и принципами его работы. Поскольку функцией этого уровня является интерпретация команд второго уровня, мы сконцентрируемся именно на этом и проиллюстрируем это на примерах. В этой главе также рассказывается о микроархитектурном уровне некоторых конкретных компьютеров.

В главе 5 обсуждается уровень архитектуры команд, который многие называют машинным языком. Здесь мы подробно рассмотрим 3 компьютера, выбранные нами в качестве иллюстративных примеров.

В главе 6 говорится о некоторых командах, об устройстве памяти компьютера, о механизмах управления на уровне операционной системы. В качестве примеров будут использованы операционные системы Windows NT, которая устанавливается на Pentium II, и UNIX, используемая на UltraSPARC II.

Глава 7 — об уровне языка ассемблера. Сюда относится и язык ассемблер, и процесс ассемблирования. Здесь также речь пойдет о компоновке.

В главе 8 обсуждаются параллельные компьютеры, важность которых возрастает с каждым днем. Одни из них действуют на основе нескольких процессоров, которые разделяют общую память, у других общей памяти нет. Одни из них представляют собой суперкомпьютеры, другие — сети рабочих станций. Все эти разновидности параллельных компьютеров будут рассмотрены подробно.

Глава 9 содержит список рекомендуемой литературы к каждому разделу, а также алфавитный список литературы, цитируемой в этой книге.

Вопросы и задания

1. Объясните следующие термины своими словами:
   1. Транслятор.
   2. Интерпретатор.
   3. Виртуальная машина.
2. Чем отличается интерпретация от трансляции?
3. Может ли компилятор производить данные сразу для микроархитектурного уровня, минуя уровень архитектуры команд? Обоснуйте все доводы за и против.
4. Можете ли вы представить многоуровневый компьютер, у которого уровень внешнего устройства и цифровой логический уровень — не самые нижние уровни? Объясните, почему.
5. Рассмотрим компьютер с идентичными интерпретаторами на первом, втором и третьем уровнях. Чтобы вызвать из памяти, определить и выполнить одну команду, интерпретатору нужно выполнить n команд. Выполнение одной команды первого уровня занимает k нс. Сколько времени будет занимать выполнение одной команды на втором, третьем и четвертом уровнях?
6. Рассмотрим многоуровневый компьютер, в котором все уровни отличаются друг от друга. Команды каждого уровня в m раз мощнее команд предыдущего уровня, то есть одна команда уровня r может выполнять ту же работу, которую выполняют m команд на уровне r–1. Если для выполнения программы первого уровня требуется k секунд, сколько времени будут выполняться соответствующие программы на втором, третьем и четвертом уровнях, учитывая, что для интерпретации одной команды уровня r+1 требуется n команд уровня r?
7. Некоторые команды уровня операционной системы идентичны командам уровня архитектуры команд. Эти команды сразу выполняются микропрограммой, а не операционной системой. Учитывая ответ на предыдущий вопрос, подумайте, зачем это нужно.
8. В каком смысле аппаратное и программное обеспечение эквивалентны? А в каком не эквивалентны?

9. Одно из следствий идеи фон Неймана хранить программу в памяти компьютера — возможность вносить изменения в программы. Приведите пример, где это может быть полезно (подсказка: подумайте об арифметических операциях над массивами).
10. Работоспособность 75-й модели IBM-360 в 50 раз больше, чем у модели 30, однако время цикла меньше всего лишь в 5 раз. Объясните, почему.
11. На рисунках 1.4 и 1.5 изображены схемы компьютерных систем. Опишите, как происходит процесс ввода-вывода в каждой из этих систем. У какой из них общая производительность больше?
12. В определенный момент времени диаметр транзистора на микропроцессоре составлял один микрон. Каков будет диаметр транзистора на новой модели в следующем году в соответствии с законом Мура?

Глава 2
Организация компьютерных систем

Цифровой компьютер состоит из связанных между собой процессоров, памяти и устройств ввода-вывода. Вторая глава знакомит читателя с этими компонентами и с тем, как они взаимосвязаны. Данная информация послужит основой для подробного рассмотрения каждого уровня в последующих пяти главах. Процессоры, память и устройства ввода-вывода — ключевые понятия, они будут упоминаться при обсуждении каждого уровня, поэтому изучение компьютерной архитектуры мы начнем с них.

Процессоры

На рис. 2.1 показано устройство обычного компьютера. **Центральный процессор** — это мозг компьютера. Его задача — выполнять программы, находящиеся в основной памяти. Он вызывает команды из памяти, определяет их тип, а затем выполняет их одну за другой. Компоненты соединены **шиной**, представляющей собой набор параллельно связанных проводов, по которым передаются адреса, данные и сигналы управления. Шины могут быть внешними (связывающими процессор с памятью и устройствами ввода-вывода) и внутренними.

Процессор состоит из нескольких частей. Блок управления отвечает за вызов команд из памяти и определение их типа. Арифметико-логическое устройство выполняет арифметические операции (например, сложение) и логические операции (например, логическое И).

Внутри центрального процессора находится память для хранения промежуточных результатов и некоторых команд управления. Эта память состоит из нескольких регистров, каждый из которых выполняет определенную функцию. Обычно все регистры одинакового размера. Каждый регистр содержит одно число, которое ограничивается размером регистра. Регистры считываются и записываются очень быстро, поскольку они находятся внутри центрального процессора.

Самый важный регистр — **счетчик команд**, который указывает, какую команду нужно выполнять дальше. Название «счетчик команд» не соответствует действительности, поскольку он ничего не считает, но этот термин употребляется повсеместно[1]. Еще есть **регистр команд**, в котором находится команда, выполняемая в данный

[1] Для этой же цели используется термин «указатель команд». — *Примеч. научн. ред.*

момент. У большинства компьютеров имеются и другие регистры, одни из них многофункциональны, другие выполняют только какие-либо специфические функции.

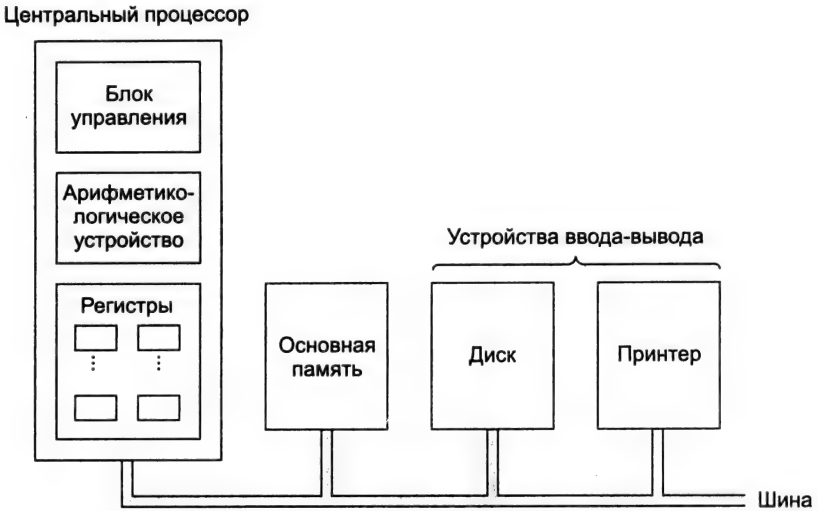


Рис. 2.1. Схема устройства компьютера с одним центральным процессором и двумя устройствами ввода-вывода

Устройство центрального процессора

Внутреннее устройство тракта данных типичного фон-неймановского процессора показано на рис. 2.2. **Тракт данных** состоит из регистров (обычно от 1 до 32), **АЛУ (арифметико-логического устройства)** и нескольких соединяющих шин. Содержимое регистров поступает во входные регистры АЛУ, которые на рис. 2.2 обозначены буквами A и B. В них находятся входные данные АЛУ, пока АЛУ производит вычисления. Тракт данных — важная составная часть всех компьютеров, и мы обсудим его очень подробно.

АЛУ выполняет сложение, вычитание и другие простые операции над входными данными и помещает результат в выходной регистр. Этот выходной регистр может помещаться обратно в один из регистров. Он может быть сохранен в памяти, если это необходимо. На рис. 2.2 показана операция сложения. Отметим, что входные и выходные регистры есть не у всех компьютеров.

Большинство команд можно разделить на две группы: команды типа регистр-память и типа регистр-регистр. Команды первого типа вызывают слова из памяти, помещают их в регистры, где они используются в качестве входных данных АЛУ. («Слова» — это такие элементы данных, которые перемещаются между памятью и регистрами[1].) Словом может быть целое число. Устройство памяти мы обсудим ниже в этой главе. Другие команды этого типа помещают регистры обратно в память.

[1] На самом деле размер слова обычно соответствует разрядности регистра данных. Так, например, у 16-битных микропроцессоров 8086 и 8088 слово было 16-битным, а у 32-битных микропроцессоров слово имеет длину 32 бита. — *Примеч. научн. ред.*

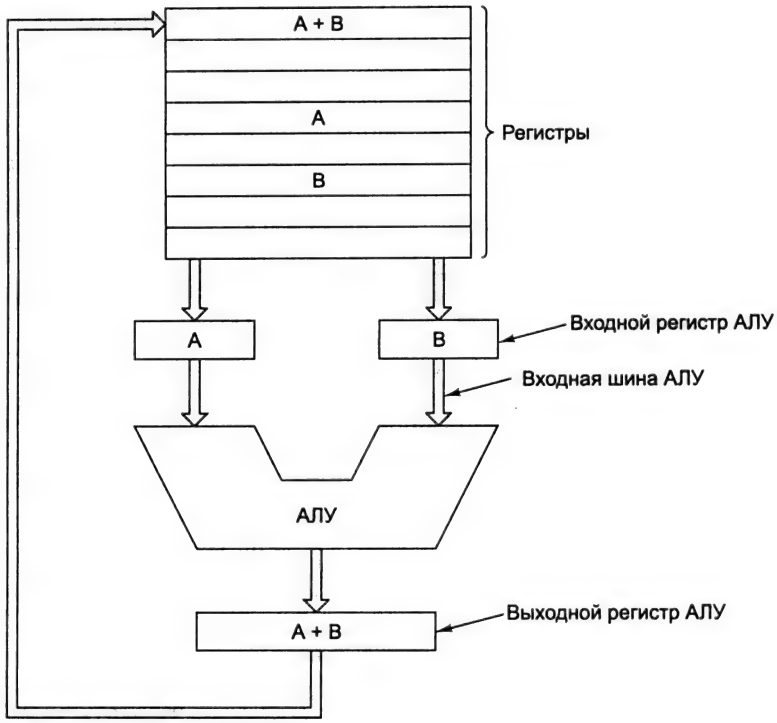


Рис. 2.2. Тракт данных в обычной фон-неймановской машине

Команды второго типа вызывают два операнда из регистров, помещают их во входные регистры АЛУ, выполняют над ними какую-нибудь арифметическую или логическую операцию и переносят результат обратно в один из регистров. Этот процесс называется **циклом тракта данных**. В какой-то степени он определяет, что может делать машина. Чем быстрее происходит цикл тракта данных, тем быстрее компьютер работает.

Выполнение команд

Центральный процессор выполняет каждую команду за несколько шагов:

1) вызывает следующую команду из памяти и переносит ее в регистр команд;
2) меняет положение счетчика команд, который теперь должен указывать на следующую команду[1];
3) определяет тип вызванной команды;
4) если команда использует слово из памяти, определяет, где находится это слово;
5) переносит слово, если это необходимо, в регистр центрального процессора[2];

[1] Это происходит после декодирования текущей команды, а иногда и после ее выполнения. — *Примеч. научн. ред.*

[2] Следует заметить, что бывают команды, которые требуют загрузки из памяти целого множества слов и их обработки в рамках одной-единственной команды. — *Примеч. научн. ред.*

6) выполняет команду;

7) переходит к шагу 1, чтобы начать выполнение следующей команды.

Такая последовательность шагов (**выборка—декодирование—исполнение**) является основой работы всех компьютеров.

Описание работы центрального процессора можно представить в виде программы на английском языке. В листинге 2.1 приведена такая программа-интерпретатор на языке Java. В описываемом компьютере есть два регистра: счетчик команд, который содержит путь к адресу следующей команды, и аккумулятор, в котором хранятся результаты арифметических операций. Кроме того, имеются внутренние регистры, в которых хранится текущая команда (`instr`), тип текущей команды (`instr_type`), адрес операнда команды (`data_loc`) и сам операнд (`data`). Каждая команда содержит один адрес ячейки памяти. В ячейке памяти находится операнд, например кусок данных, который нужно добавить в аккумулятор.

Листинг 2.1. Интерпретатор для простого компьютера (на языке Java)

```
public class Interp{
    static int PC;              //PC содержит адрес следующей команды
    static int AC;              // аккумулятор, регистр для арифметики
    static int instr;           //регистр для текущей команды
    static int instr_type;      //тип команды (код операции)
    static int data_loc;        //адрес данных или -1, если его нет
    static int data;            //содержит текущий операнд
    static boolean run_bit = true;  //бит, который можно выключить, чтобы остановить машину

                                public static void interpret(int memory[], int starting_address{
                                //Эта процедура интепретирует программы для простой машины,
которая содержит команды только с одним операндом из памяти. Машина содержит регистр AC
(аккумулятор). Он используется для арифметических действий. Например, команда ADD суммирует
число из памяти с AC. Интерпретатор работает до тех пор, пока не будет выполнена команда
HALT, вследствие чего бит run_bit поменяет значение на false. Машина состоит из памяти,
счетчика команд, бита run bit и аккумулятора AC. Входные параметры состоят из копии
содержимого памяти и начального адреса.

        PC=starting_address;
        while (run_bit) {
            instr=memory[PC]; //вызывает следующую команду в instr
            PC=PC+1;          //увеличивает значение счетчика команд
            instr_type=get_instr_type(instr);       //определяет тип команды
            data_loc=find_data(instr, instr_type); //находит данные (-1, если данных нет)
            if(data_loc>=0)   //если data_lock=-1, //значит, операнда нет
                data=memory[data_loc];   //вызов данных
            execute(instr_type,data);    //выполнение команды
        }
    }

    private static int get_instr_type(int addr) {…}
    private static int find_data(int instr, int type) {…}
    private static void execute(int type, int data) {…}

}
```

Сама возможность написать программу, имитирующей работу центрального процессора, показывает, что программа не обязательно должна выполняться реальным процессором, относящимся к аппаратному обеспечению. Напротив, вызывать

из памяти, определять тип команд и выполнять эти команды может другая программа. Такая программа называется **интерпретатором**. Об интерпретаторах мы говорили в главе 1.

Написание программ-интерпретаторов, которые имитируют работу процессора, широко используется при разработке компьютерных систем. После того как разработчики выбрали машинный язык (Я) для нового компьютера, они должны решить, строить ли им процессор, который будет выполнять программы на языке Я, или написать специальную программу для интерпретации программ на языке Я. Если они решают написать интерпретатор, они должны создать аппаратное обеспечение для выполнения этого интерпретатора. Возможны также гибридные конструкции, когда часть команд выполняется аппаратным обеспечением, а часть интерпретируется.

Интерпретатор разбивает команды на маленькие шаги. Таким образом, машина с интерпретатором может быть гораздо проще по строению и дешевле, чем процессор, выполняющий программы без интерпретации. Такая экономия особенно важна, если компьютер содержит большое количество сложных команд с различными опциями. В сущности, экономия проистекает из самой замены аппаратного обеспечения программным обеспечением (интерпретатором).

Первые компьютеры содержали небольшое количество команд, и эти команды были простыми. Но поиски более мощных компьютеров привели, кроме всего прочего, к появлению более сложных команд. Вскоре разработчики поняли, что при наличии сложных команд программы выполняются быстрее, хотя выполнение отдельных команд занимает больше времени. В качестве примеров сложных команд можно назвать выполнение операций с плавающей точкой, обеспечение прямого доступа к элементам массива и т. п. Если обнаруживалось, что две определенные команды часто выполнялись последовательно одна за другой, то вводилась новая команда, заменяющая работу этих двух.

Сложные команды были лучше, потому что некоторые операции иногда перекрывались. Какие-то операции могли выполняться параллельно, для этого использовались разные части аппаратного обеспечения. Для дорогих компьютеров с высокой производительностью стоимость этого дополнительного аппаратного обеспечения была вполне оправданна. Таким образом, у дорогих компьютеров было гораздо больше команд, чем у дешевых. Однако развитие программного обеспечения и требования совместимости команд привели к тому, что сложные команды стали использоваться и в дешевых компьютерах, хотя там во главу угла ставилась стоимость, а не скорость работы.

К концу 50-х годов компания IBM, которая лидировала тогда на компьютерном рынке, решила, что производство семейства компьютеров, каждый из которых выполняет одни и те же команды, имеет много преимуществ и для самой компании, и для покупателей. Чтобы описать этот уровень совместимости, компания IBM ввела термин **архитектура**. Новое семейство компьютеров должно было иметь одну общую архитектуру и много разных разработок, различающихся по цене и скорости, которые могли выполнять одну и ту же программу. Но как построить дешевый компьютер, который будет выполнять все сложные команды, предназначенные для высокоэффективных дорогостоящих машин?

Решением этой проблемы стала интерпретация. Эта технология, впервые предложенная Уилксом в 1951 году, позволяла разрабатывать простые дешевые компьютеры, которые, тем не менее, могли выполнять большое количество команд. В результате IBM создала архитектуру System/360, семейство совместимых компьютеров, различных по цене и производительности. Аппаратное обеспечение без интерпретации использовалось только в самых дорогих моделях.

Простые компьютеры с интерпретированными командами имели некоторые другие преимущества. Наиболее важными среди них были:

1) возможность фиксировать неправильно выполненные команды или даже восполнять недостатки аппаратного обеспечения;
2) возможность добавлять новые команды при минимальных затратах, даже после покупки компьютера;
3) структурированная организация, которая позволяла разрабатывать, проверять и документировать сложные команды.

В 70-е годы компьютерный рынок быстро разрастался, новые компьютеры могли выполнять все больше и больше функций. Спрос на дешевые компьютеры провоцировал создание компьютеров с использованием интерпретаторов. Возможность разрабатывать аппаратное обеспечение и интерпретатор для определенного набора команд вылилась в создание дешевых процессоров. Полупроводниковые технологии быстро развивались, преимущества низкой стоимости преобладали над возможностями более высокой производительности, и использование интерпретаторов при разработке компьютеров стало широко применимо. Интерпретация использовалась практически во всех компьютерах, выпущенных в 70-е годы, от миникомпьютеров до самых больших машин.

К концу 70-х годов интерпретаторы стали применяться практически во всех моделях, кроме самых дорогостоящих машин с очень высокой производительностью (например, Cray-1 и компьютеров серии Control Data Cyber). Использование интерпретаторов исключало высокую стоимость сложных команд, и разработчики могли вводить все более и более сложные команды, в особенности различные способы определения используемых операндов.

Эта тенденция достигла пика своего развития в разработке компьютера VAX (производитель Digital Equipment Corporation), у которого было несколько сотен команд и более 200 способов определения операндов в каждой команде. К несчастью, архитектура VAX с самого начала разрабатывалась с использованием интерпретатора, а производительности уделялось мало внимания. Это привело к появлению большого количества команд второстепенного значения, которые трудно было выполнять сразу без интерпретации. Данное упущение стало фатальным как для VAX, так и для его производителя (компании DEC). Compaq купил DEC в 1998 году.

Хотя самые первые 8-битные микропроцессоры были очень простыми и содержали небольшой набор команд, к концу 70-х годов даже они стали разрабатываться с использованием интерпретаторов. В этот период основной проблемой для разработчиков стала возрастающая сложность микропроцессоров. Главное преимущество интерпретации заключалось в том, что можно было разработать простой процессор, а вся сложность сводилась к созданию интерпретатора. Таким образом,

разработка сложного аппаратного обеспечения замещалась разработкой сложного программного обеспечения.

Успех Motorola 68000 с большим набором интерпретируемых команд и одновременный провал Zilog Z8000, у которого был столь же обширный набор команд, но не было интерпретатора, продемонстрировали все преимущества использования интерпретаторов при разработке новых машин. Успех Motorola 68000 был несколько неожиданным, учитывая, что Z80 (предшественник Zilog Z8000) пользовался большей популярностью, чем Motorola 6800 (предшественник Motorola 68000). Конечно, важную роль здесь играли и другие факторы, например то, что Motorola много лет занималась производством микросхем, а Exxon (владелец Zilog) долгое время был нефтяной компанией.

Еще один фактор в пользу интерпретации — существование быстрых постоянных запоминающих устройств (так называемых командных ПЗУ) для хранения интерпретаторов. Предположим, что для выполнения обычной интерпретируемой команды Motorola 68000 интерпретатору нужно выполнить 10 команд, которые называются **микрокомандами**, по 100 нс каждая, и произвести 2 обращения к оперативной памяти по 500 нс каждое. Общее время выполнения команды составит, следовательно, 2000 нс, всего лишь в два раза больше, чем в лучшем случае могло бы занять непосредственное выполнение этой команды без интерпретации. А если бы не было специального быстродействующего постоянного запоминающего устройства, выполнение этой команды заняло бы целых 6000 нс. Таким образом, важность наличия командных ПЗУ очевидна.

RISC и CISC

В конце 70-х годов проводилось много экспериментов с очень сложными командами, появление которых стало возможным благодаря интерпретации. Разработчики пытались уменьшить пропасть между тем, что компьютеры способны делать, и тем, что требуют языки высокого уровня. Едва ли кто-нибудь тогда думал о разработке более простых машин, так же как сейчас мало кто занимается разработкой менее мощных операционных систем, сетей, редакторов и т. д. (к несчастью).

В компании IBM группа разработчиков во главе с Джоном Коком противостояла этой тенденции; они попытались воплотить идеи Сеймура Крея, создав экспериментальный высокоэффективный мини-компьютер **801**. Хотя IBM не занималась сбытом этой машины, а результаты эксперимента были опубликованы только через несколько лет, весть быстро разнеслась по свету, и другие производители тоже занялись разработкой подобных архитектур.

В 1980 году группа разработчиков в университете Беркли во главе с Дэвидом Паттерсоном и Карло Секвином начала разработку процессоров VLSI без использования интерпретации. Для обозначения этого понятия они придумали термин **RISC** и назвали новый процессор RISC I, вслед за которым вскоре был выпущен RISC II. Немного позже, в 1981 году, Джон Хеннеси в Стенфорде разработал и выпустил другую микросхему, которую он назвал **MIPS**. Эти две микросхемы развились в коммерчески важные продукты SPARC и MIPS соответственно.

Новые процессоры существенно отличались от коммерческих процессоров того времени. Поскольку они не были совместимы с существующей продукцией, раз-

работчики вправе были включать туда новые наборы команд, которые могли бы увеличить общую производительность системы. Так как основное внимание уделялось простым командам, которые могли быстро выполняться, разработчики вскоре осознали, что ключом к высокой производительности компьютера была разработка команд, к выполнению которых можно быстро приступать. Сколько времени занимает выполнение одной команды, было не так важно, как то, сколько команд может быть начато в секунду.

В то время как разрабатывались эти простые процессоры, всеобщее внимание привлекало относительно небольшое количество команд (обычно их было около 50). Для сравнения: число команд в DEC VAX и больших IBM в то время составляло от 200 до 300. RISC — это сокращение от **Reduced Instruction Set Computer — компьютер с сокращенным набором команд**. RISC противопоставлялся **CISC (Complex Instruction Set Computer — компьютер с полным набором команд)**. В качестве примера CISC можно привести VAX, который доминировал в то время в научных компьютерных центрах. На сегодняшний день мало кто считает, что главное различие RISC и CISC состоит в количестве команд, но название сохраняется до сих пор.

С этого момента началась грандиозная идеологическая война между сторонниками RISC и разработчиками VAX, Intel и больших IBM. По их мнению, наилучший способ разработки компьютеров — включение туда небольшого количества простых команд, каждая из которых выполняется за один цикл тракта данных (см. рис. 2.2), то есть берет два регистра, производит над ними какую-либо арифметическую или логическую операцию (например, сложения или логическое И) и помещает результат обратно в регистр. В качестве аргумента они утверждали, что даже если RISC должна выполнять 4 или 5 команд вместо одной, которую выполняет CISC, притом что команды RISC выполняются в 10 раз быстрее (поскольку они не интерпретируются), он выигрывает в скорости. Следует также отметить, что к этому времени скорость работы основной памяти приблизилась к скорости специальных управляющих постоянных запоминающих устройств, потому недостатки интерпретации были налицо, что повышало популярность компьютеров RISC.

Учитывая преимущества производительности RISC, можно было бы предположить, что такие компьютеры, как Alpha компании DEC, стали доминировать над компьютерами CISC (Pentium и т. д.) на рынке. Однако ничего подобного не произошло. Возникает вопрос: почему?

Во-первых, компьютеры RISC были несовместимы с другими моделями, а многие компании вложили миллиарды долларов в программное обеспечение для продукции Intel. Во-вторых, как ни странно, компания Intel сумела воплотить те же идеи в архитектуре CISC. Процессоры Intel, начиная с 486-го, содержат ядро RISC, которое выполняет самые простые (и обычно самые распространенные) команды за один цикл тракта данных, а по обычной технологии CISC интерпретируются более сложные команды. В результате обычные команды выполняются быстро, а более сложные и редкие — медленно. Хотя при таком «гибридном» подходе работа происходит не так быстро, как у RISC, данная архитектура имеет ряд преимуществ, поскольку позволяет использовать старое программное обеспечение без изменений.

Принципы разработки современных компьютеров

Прошло уже более двадцати лет с тех пор, как были сконструированы первые компьютеры RISC, однако некоторые принципы разработки можно перенять, учитывая современное состояние технологий аппаратного обеспечения. Если происходит очень резкое изменение в технологиях (например, новый процесс производства делает время цикла памяти в 10 раз меньше, чем время цикла центрального процессора), меняются все условия. Поэтому разработчики всегда должны учитывать возможные технологические изменения, которые могут повлиять на баланс между компонентами компьютера.

Существует ряд принципов разработки, иногда называемых **принципами RISC**, которым по возможности стараются следовать производители универсальных процессоров. Из-за некоторых внешних ограничений, например требования совместимости с другими машинами, приходится время от времени идти на компромисс, но эти принципы — цель, к которой стремится большинство разработчиков. Ниже мы обсудим некоторые из них.

Все команды непосредственно выполняются аппаратным обеспечением. Все обычные команды непосредственно выполняются аппаратным обеспечением. Они не интерпретируются микрокомандами. Устранение уровня интерпретации обеспечивает высокую скорость выполнения большинства команд. В компьютерах типа CISC более сложные команды могут разбиваться на несколько частей, которые затем выполняются как последовательность микрокоманд. Эта дополнительная операция снижает скорость работы машины, но она может быть применима для редко встречающихся команд.

Компьютер должен начинать выполнение большого числа команд. В современных компьютерах используется много различных способов для увеличения производительности, главное из которых — возможность обращаться к как можно большему количеству команд в секунду. Процессор 500-MIPS способен приступать к выполнению 500 млн команд в секунду, и при этом не имеет значения, сколько времени занимает само выполнение этих команд. (**MIPS** — это сокращение от **Millions of Instructions Per Second** — **«миллионы команд в секунду»**.) Этот принцип предполагает, что параллелизм может играть главную роль в улучшении производительности, поскольку приступать к большому количеству команд за короткий промежуток времени можно только в том случае, если одновременно может выполняться несколько команд.

Хотя команды некоторой программы всегда расположены в определенном порядке, компьютер может приступать к их выполнению и в другом порядке (так как необходимые ресурсы памяти могут быть заняты) и, кроме того, может заканчивать их выполнение не в том порядке, в котором они расположены в программе. Конечно, если команда 1 устанавливает регистр, а команда 2 использует этот регистр, нужно действовать с особой осторожностью, чтобы команда 2 не считывала регистр до тех пор, пока он не будет содержать нужное значение. Чтобы не допустить подобных ошибок, необходимо вводить большое количество соответствующих записей в память, но производительность все равно становится выше благодаря возможности выполнять несколько команд одновременно.

Команды должны легко декодироваться. Предел количества вызываемых команд в секунду зависит от процесса декодирования отдельных команд. Декодирование команд осуществляется для того, чтобы определить, какие ресурсы им необходимы и какие действия нужно выполнить. Полезны любые средства, которые способствуют упрощению этого процесса. Например, используются регулярные команды с фиксированной длиной и с небольшим количеством полей. Чем меньше разных форматов команд, тем лучше.

К памяти должны обращаться только команды загрузки и сохранения. Один из самых простых способов разбивания операций на отдельные шаги — потребовать, чтобы операнды для большинства команд брались из регистров и возвращались туда же. Операция перемещения операндов из памяти в регистры может осуществляться в разных командах. Поскольку доступ к памяти занимает много времени, а подобная задержка нежелательна, работу этих команд могут выполнять другие команды, если они не делают ничего, кроме передвижения операндов между регистрами и памятью. Из этого наблюдения следует, что к памяти должны обращаться только команды загрузки и сохранения (LOAD и STORE).

Должно быть большое количество регистров. Поскольку доступ к памяти происходит довольно медленно, в компьютере должно быть много регистров (по крайней мере 32). Если слово однажды вызвано из памяти, при наличии большого числа регистров оно может содержаться в регистре до тех пор, пока будет не нужно. Возвращение слова из регистра в память и новая загрузка этого же слова в регистр нежелательны. Лучший способ избежать излишних перемещений — наличие достаточного количества регистров.

Параллелизм на уровне команд

Разработчики компьютеров стремятся к тому, чтобы улучшить производительность машин, над которыми они работают. Один из способов заставить процессоры работать быстрее — увеличение их скорости, однако при этом существуют некоторые технологические ограничения, связанные с конкретным историческим периодом. Поэтому большинство разработчиков для достижения лучшей производительности при данной скорости работы процессора используют параллелизм (возможность выполнять две или более операций одновременно).

Существует две основные формы параллелизма: параллелизм на уровне команд и параллелизм на уровне процессоров. В первом случае параллелизм осуществляется в пределах отдельных команд и обеспечивает выполнение большого количества команд в секунду. Во втором случае над одной задачей работают одновременно несколько процессоров. Каждый подход имеет свои преимущества. В этом разделе мы рассмотрим параллелизм на уровне команд, а в следующем — параллелизм на уровне процессоров.

Конвейеры

Уже много лет известно, что главным препятствием высокой скорости выполнения команд является их вызов из памяти. Для разрешения этой проблемы разработчики придумали средство для вызова команд из памяти заранее, чтобы они

имелись в наличии в тот момент, когда будут необходимы. Эти команды помещались в набор регистров, который назывался **буфером выборки с упреждением**. Таким образом, когда была нужна определенная команда, она вызывалась прямо из буфера, и не нужно было ждать, пока она считается из памяти. Эта идея использовалась еще при разработке IBM Stretch, который был сконструирован в 1959 году.

В действительности процесс выборки с упреждением подразделяет выполнение команды на два этапа: вызов и собственно выполнение. Идея **конвейера** еще больше продвинула эту стратегию вперед. Теперь команда подразделялась уже не на два, а на несколько этапов, каждый из которых выполнялся определенной частью аппаратного обеспечения, причем все эти части могли работать параллельно.

На рис. 2.3, *а* изображен конвейер из 5 блоков, которые называются **стадиями**. Стадия С1 вызывает команду из памяти и помещает ее в буфер, где она хранится до тех пор, пока не будет нужна. Стадия С2 декодирует эту команду, определяя ее тип и тип операндов, над которыми она будет производить определенные действия. Стадия С3 определяет местонахождение операндов и вызывает их или из регистров, или из памяти. Стадия С4 выполняет команду, обычно путем провода операндов через тракт данных (см. рис. 2.2). И наконец, стадия С5 записывает результат обратно в нужный регистр.

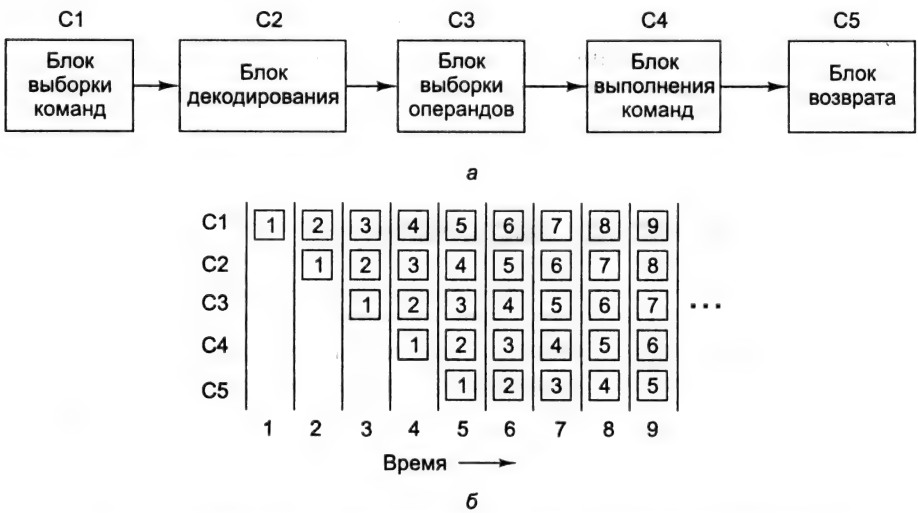


Рис. 2.3. Конвейер из 5 стадий (*а*); состояние каждой стадии в зависимости от количества пройденных циклов (*б*). Показано 9 циклов

На рис. 2.3, *б* мы видим, как действует конвейер во времени. Во время цикла 1 стадия С1 работает над командой 1, вызывая ее из памяти. Во время цикла 2 стадия С2 декодирует команду 1, в то время как стадия С1 вызывает из памяти команду 2. Во время цикла 3 стадия С3 вызывает операнды для команды 1, стадия С2 декодирует команду 2, а стадия С1 вызывает третью команду. Во время цикла 4 стадия С4 выполняет команду 1, С3 вызывает операнды для команды 2, С2 декодирует команду 3, а С1 вызывает команду 4. Наконец, во время пятого цикла С5 записывает результат выполнения команды 1 обратно в регистр, тогда как другие стадии работают над следующими командами.

Чтобы лучше понять принципы работы конвейера, рассмотрим аналогичный пример. Представим себе кондитерскую фабрику, на которой выпечка тортов и их упаковка для отправки производятся раздельно. Предположим, что в отделе отправки находится длинный конвейер, вдоль которого стоят 5 рабочих (или блоков обработки). Каждые 10 секунд (это время цикла) первый рабочий ставит пустую коробку для торта на ленту конвейера. Эта коробка отправляется ко второму рабочему, который кладет в нее торт. После этого коробка с тортом доставляется третьему рабочему, который закрывает и запечатывает ее. Затем она поступает к четвертому рабочему, который ставит на ней ярлык. Наконец, пятый рабочий снимает коробку с конвейерной ленты и помещает ее в большой контейнер для отправки в супермаркет. Примерно таким же образом действует компьютерный конвейер: каждая команда (в случае с кондитерской фабрикой — торт) перед окончательным выполнением проходит несколько шагов обработки.

Возвратимся к нашему конвейеру, изображенному на рис. 2.3. Предположим, что время цикла у этой машины 2 нс. Тогда для того, чтобы одна команда прошла через весь конвейер, требуется 10 нс. На первый взгляд может показаться, что такой компьютер может выполнять 100 млн команд в секунду, в действительности же скорость его работы гораздо выше. Во время каждого цикла (2 нс) завершается выполнение одной новой команды, поэтому машина выполняет не 100 млн, а 500 млн команд в секунду.

Конвейеры позволяют найти компромисс между **временем ожидания** (сколько времени занимает выполнение одной команды) и **пропускной способностью процессора** (сколько миллионов команд в секунду выполняет процессор). Если время цикла составляет T нс, а конвейер содержит n стадий, то время ожидания составит nT нс, а пропускная способность — 1000/T млн команд в секунду.

Суперскалярные архитектуры

Один конвейер — хорошо, а два — еще лучше. Одна из возможных схем процессора с двойным конвейером показана на рис. 2.4. В основе разработки лежит конвейер, изображенный на рис. 2.3. Здесь общий отдел вызова команд берет из памяти сразу по две команды и помещает каждую из них в один из конвейеров. Каждый конвейер содержит АЛУ для параллельных операций. Чтобы выполняться параллельно, две команды не должны конфликтовать при использовании ресурсов (например, регистров), и ни одна из них не должна зависеть от результата выполнения другой. Как и в случае с одним конвейером, либо компилятор должен следить, чтобы не возникало неприятных ситуаций (например, когда аппаратное обеспечение выдает некорректные результаты, если команды несовместимы), либо же конфликты выявляются и устраняются прямо во время выполнения команд благодаря использованию дополнительного аппаратного обеспечения.

Сначала конвейеры (как двойные, так и одинарные) использовались только в компьютерах RISC. У 386-го и его предшественников их не было. Конвейеры в процессорах компании Intel появились только начиная с 486-й модели[1]. 486-й процес-

[1] Необходимо отметить, что параллельное функционирование отдельных блоков процессора использовалось и в предыдущем — 386-м — микропроцессоре. Оно стало прообразом 5-стадийного конвейера микропроцессора 486. — *Примеч. научн. ред.*

сор содержал один конвейер, а Pentium — два конвейера из пяти стадий. Похожая схема изображена на рис. 2.4, но разделение функций между второй и третьей стадиями (они назывались декодирование 1 и декодирование 2) было немного другим. Главный конвейер (**u-конвейер**) мог выполнять произвольные команды. Второй конвейер (**v-конвейер**) мог выполнять только простые команды с целыми числами, а также одну простую команду с плавающей точкой (FXCH).

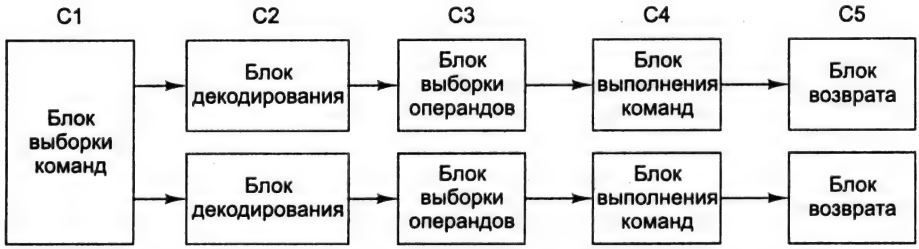


Рис. 2.4. Двойной конвейер из пяти стадий с общим отделом вызова команд

Имеются сложные правила определения, является ли пара команд совместимой для того, чтобы выполняться параллельно. Если команды, входящие в пару, были сложными или несовместимыми, выполнялась только одна из них (в u-конвейере). Оставшаяся вторая команда составляла затем пару со следующей командой. Команды всегда выполнялись по порядку. Таким образом, Pentium содержал особые компиляторы, которые объединяли совместимые команды в пары и могли порождать программы, выполняющиеся быстрее, чем в предыдущих версиях. Измерения показали, что программы, производящие операции с целыми числами, на компьютере Pentium выполняются почти в два раза быстрее, чем на 486-м, хотя у него такая же тактовая частота. Вне всяких сомнений, преимущество в скорости появилось благодаря второму конвейеру.

Переход к четырем конвейерам возможен, но это потребовало бы создания громоздкого аппаратного обеспечения (отметим, что компьютерщики, в отличие от фольклористов, не верят в счастливое число три). Вместо этого используется другой подход. Основная идея — один конвейер с большим количеством функциональных блоков, как показано на рис. 2.5. Pentium II, к примеру, имеет сходную структуру (подробно мы рассмотрим его в главе 4). В 1987 году для обозначения этого подхода был введен термин **суперскалярная архитектура**. Однако подобная идея нашла воплощение еще более 30 лет назад в компьютере CDC 6600. CDC 6600 вызывал команду из памяти каждые 100 нс и помещал ее в один из 10 функциональных блоков для параллельного выполнения. Пока команды выполнялись, центральный процессор вызывал следующую команду.

Отметим, что стадия 3 выпускает команды значительно быстрее, чем стадия 4 способна их выполнять. Если бы стадия 3 выпускала команду каждые 10 нс, а все функциональные блоки выполняли бы свою работу также за 10 нс, то на четвертой стадии всегда функционировал бы только один блок, что сделало бы саму идею конвейера бессмысленной. В действительности большинству функциональных блоков четвертой стадии для выполнения команды требуется значительно больше

времени, чем занимает один цикл (это блоки доступа к памяти и блок выполнения операций с плавающей точкой). Как видно из рис. 2.5, на четвертой стадии может быть несколько АЛУ.

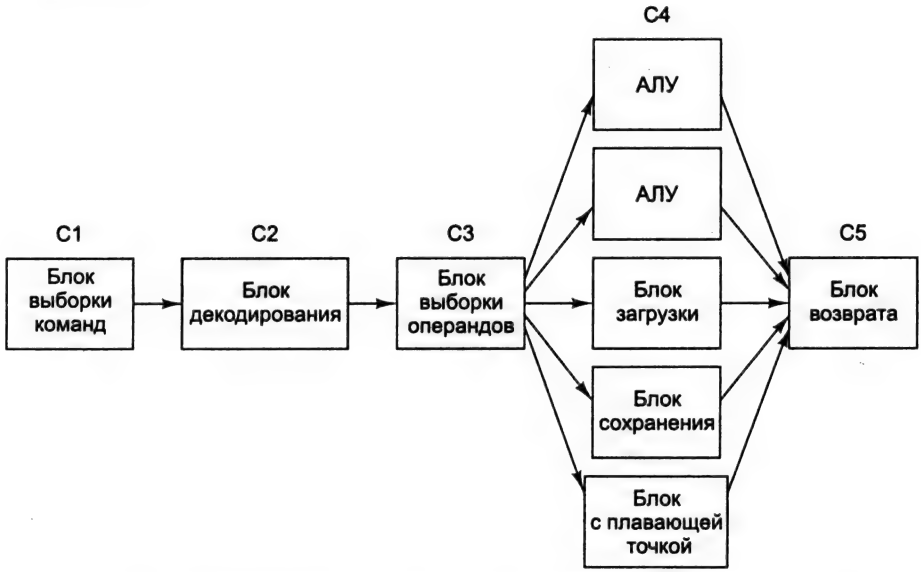


Рис. 2.5. Суперскалярный процессор с пятью функциональными блоками

Параллелизм на уровне процессоров

Спрос на компьютеры, работающие все с более и более высокой скоростью, не прекращается. Астрономы хотят выяснить, что произошло в первую микросекунду после большого взрыва, экономисты хотят смоделировать всю мировую экономику, подростки хотят играть в 3D интерактивные игры со своими виртуальными друзьями через Интернет. Скорость работы процессоров повышается, но у них постоянно возникают проблемы с быстротой передачи информации, поскольку скорость распространения электромагнитных волн в медных проводах и света в оптико-волоконных кабелях по-прежнему остается 20 см/нс, независимо от того, насколько умны инженеры компании Intel. Кроме того, чем быстрее работает процессор, тем сильнее он нагревается[1], и нужно предохранять его от перегрева.

Параллелизм на уровне команд помогает в какой-то степени, но конвейеры и суперскалярная архитектура обычно увеличивают скорость работы всего лишь в 5–10 раз. Чтобы улучшить производительность в 50, 100 и более раз, нужно разрабатывать компьютеры с несколькими процессорами. Ниже мы ознакомимся с устройством таких компьютеров.

[1] Это не совсем точно. Есть масса живых примеров, противоречащих этому высказыванию. Тепловыделение, конечно, зависит от частоты переключений элементов, но оно зависит и от размеров этих элементов, и от напряжения, от которого они работают. — *Примеч. научн. ред.*

Векторные компьютеры

Многие задачи в физических и технических науках содержат векторы, в противном случае они имели бы очень сложную структуру. Часто одни и те же вычисления выполняются над разными наборами данных в одно и то же время. Структура этих программ позволяет повышать скорость работы благодаря параллельному выполнению команд. Существует два метода, которые используются для быстрого выполнения больших научных программ. Хотя обе схемы во многих отношениях схожи, одна из них считается расширением одного процессора, а другая — параллельным компьютером.

Массивно-параллельный процессор (array processor) состоит из большого числа сходных процессоров, которые выполняют одну и ту же последовательность команд применительно к разным наборам данных. Первым в мире таким процессором был ILLIAC IV (Университет Иллинойса). Он изображен на рис. 2.6. Первоначально предполагалось сконструировать машину, состоящую из четырех секторов, каждый из которых содержит решетку 8×8 элементов процессор/память. Для каждого сектора имелся один блок контроля. Он рассылал команды, которые выполнялись всеми процессорами одновременно, при этом каждый процессор использовал свои собственные данные из своей собственной памяти (загрузка данных происходила во время инициализации). Из-за очень высокой стоимости был построен только один такой сектор, но он мог выполнять 50 млн операций с плавающей точкой в секунду. Если бы при создании машины использовалось четыре сектора и она могла бы выполнять 1 млрд операций с плавающей точкой в секунду, то мощность такой машины в два раза превышала бы мощность компьютеров всего мира.

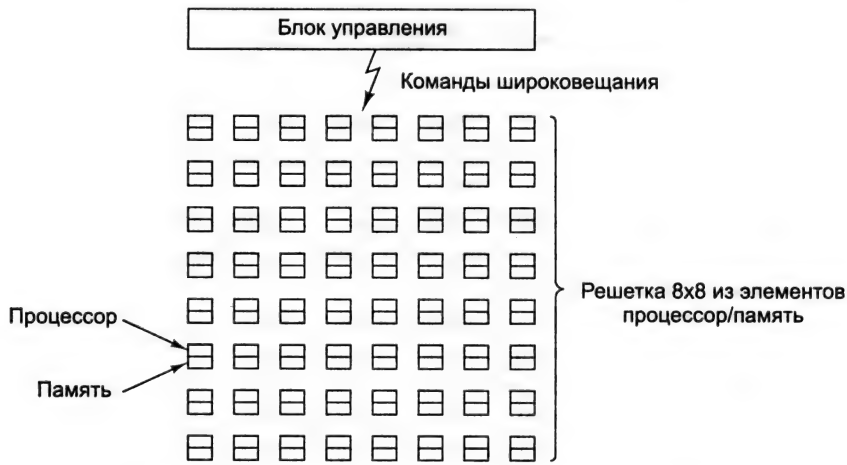


Рис. 2.6. Массивно-параллельный процессор ILLIAC IV

Для программистов **векторный процессор (vector processor)** очень похож на массивно-параллельный процессор (array processor). Как и массивно-параллельный процессор, он очень эффективен при выполнении последовательности операций над парами элементов данных. Но, в отличие от первого (array processor), все операции сложения выполняются в одном блоке суммирования, который имеет

конвейерную структуру. Компания Cray Research, основателем которой был Сеймур Крей, выпустила много векторных процессоров, начиная с модели Cray-1 (1974) и по сей день. Cray Research в настоящее время входит в состав SGI.

Оба типа процессоров работают с массивами данных. Оба они выполняют одни и те же команды, которые, например, попарно складывают элементы для двух векторов. Но если у массивно-параллельного процессора (array processor) есть столько же суммирующих устройств, сколько элементов в массиве, векторный процессор (vector processor) содержит **векторный регистр**, который состоит из набора стандартных регистров. Эти регистры последовательно загружаются из памяти при помощи одной команды. Команда сложения попарно складывает элементы двух таких векторов, загружая их из двух векторных регистров в суммирующее устройство с конвейерной структурой. В результате из суммирующего устройства выходит другой вектор, который или помещается в векторный регистр, или сразу используется в качестве операнда при выполнении другой операции с векторами.

Массивно-параллельные процессоры (array processor) выпускаются до сих пор, но занимают незначительную сферу компьютерного рынка, поскольку они эффективны при решении только таких задач, которые требуют одновременного выполнения одних и тех же вычислений над разными наборами данных. Массивно-параллельные процессоры (array processor) могут выполнять некоторые операции гораздо быстрее, чем векторные компьютеры (vector computer), но они требуют большего количества аппаратного обеспечения, и для них сложно писать программы. Векторный процессор (vector processor), с другой стороны, можно добавлять к обычному процессору. В результате те части программы, которые могут быть преобразованы в векторную форму, выполняются векторным блоком, а остальная часть программы — обычным процессором.

Мультипроцессоры

Элементы массивно-параллельного процессора связаны между собой, поскольку их работу контролирует один блок управления. Система нескольких параллельных процессоров, разделяющих общую память, называется **мультипроцессором**. Поскольку каждый процессор может записывать или считывать информацию из любой части памяти, их работа должна согласовываться программным обеспечением, чтобы не допустить каких-либо пересечений.

Возможны разные способы воплощения этой идеи. Самый простой из них — наличие одной шины, соединяющей несколько процессоров и одну общую память. Схема такого мультипроцессора показана на рис. 2.7, *а*. Такие системы производят многие компании.

Нетрудно понять, что при наличии большого числа быстро работающих процессоров, которые постоянно пытаются получить доступ к памяти через одну и ту же шину, будут возникать конфликты. Чтобы разрешить эту проблему и повысить производительность компьютера, были разработаны различные модели. Одна из них изображена на рис. 2.7, *б*. В таком компьютере каждый процессор имеет свою собственную локальную память, которая недоступна для других процессоров. Эта память используется для программ и данных, которые не нужно разделять между несколькими процессорами. При доступе к локальной памяти главная шина не используется, и, таким образом, поток информации в этой шине снижается. Возможны и другие варианты решения проблемы (например, кэш-память).

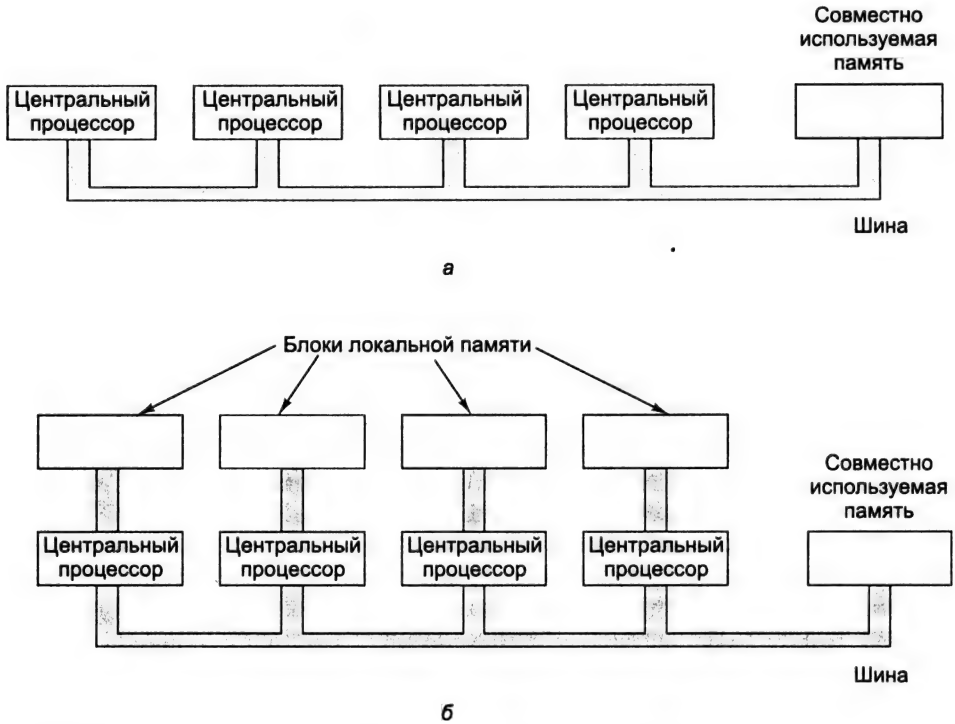


Рис. 2.7. Мультипроцессор с одной шиной и одной общей памятью (*а*); мультипроцессор, в котором для каждого процессора имеется собственная локальная память (*б*)

Мультипроцессоры имеют преимущество перед другими видами параллельных компьютеров, поскольку с единой разделенной памятью очень легко работать. Например, представим, что программа ищет раковые клетки на сделанном через микроскоп снимке ткани. Фотография в цифровом виде может храниться в общей памяти, при этом каждый процессор обследует какую-нибудь определенную область фотографии. Поскольку каждый процессор имеет доступ к общей памяти, обследование клетки, которая начинается в одной области и продолжается в другой, не представляет трудностей.

Мультикомпьютеры

Мультипроцессоры с небольшим числом процессоров (≤ 64) сконструировать довольно легко, а вот создание больших мультипроцессоров представляет некоторые трудности. Сложность заключается в том, чтобы связать все процессоры с памятью. Чтобы избежать таких проблем, многие разработчики просто отказались от идеи разделенной памяти и стали создавать системы, состоящие из большого числа взаимосвязанных компьютеров, у каждого из которых имеется своя собственная память, а общей памяти нет. Такие системы называются **мультикомпьютерами**.

Процессоры мультикомпьютера отправляют друг другу послания (это несколько похоже на электронную почту, но гораздо быстрее). Каждый компьютер не обязательно связывать со всеми другими, поэтому обычно в качестве топологий используются

2D, 3D, деревья и кольца. Чтобы послания могли дойти до места назначения, они должны проходить через один или несколько промежуточных компьютеров. Тем не менее время передачи занимает всего несколько микросекунд. Сейчас создаются и запускаются в работу мультикомпьютеры, содержащие около 10 000 процессоров.

Поскольку мультипроцессоры легче программировать, а мультикомпьютеры — конструировать, появилась идея создания гибридных систем, которые сочетают в себе преимущества обоих видов машин. Такие компьютеры представляют иллюзию разделенной памяти, при этом в действительности она не конструируется и не требует особых денежных затрат. Мы рассмотрим мультипроцессоры и мультикомпьютеры подробнее в главе 8.

Основная память

Память — часть компьютера, где хранятся программы и данные. Можно также употреблять термин «запоминающее устройство». Без памяти, откуда процессоры считывают и куда записывают информацию, не было бы цифровых компьютеров со встроенными программами.

Бит

Основной единицей памяти является двоичный разряд, который называется **битом**. Бит может содержать 0 или 1. Эта самая маленькая единица памяти. (Устройство, в котором хранятся только нули, вряд ли могло быть основой памяти. Необходимы по крайней мере две величины.)

Многие полагают, что в компьютерах используется бинарная арифметика, потому что это «эффективно». Они имеют в виду (хотя сами это редко осознают), что цифровая информация может храниться благодаря различию между разными величинами какой-либо физической характеристики, например напряжения или тока. Чем больше величин, которые нужно различать, тем меньше различий между смежными величинами и тем менее надежна память. Двоичная система требует различения всего двух величин, следовательно, это самый надежный метод кодирования цифровой информации. Если вы не знакомы с двоичной системой счисления, смотрите Приложение А.

Считается, что некоторые компьютеры, например большие IBM, используют и десятичную, и двоичную арифметику. На самом деле здесь применяется так называемый **двоично-десятичный код**. Для хранения одного десятичного разряда используется 4 бита. Эти 4 бита дают 16 комбинаций для размещения 10 различных значений (от 0 до 9). При этом 6 оставшихся комбинаций не используются. Ниже показано число 1944 в двоично-десятичной и чисто двоичной системах счисления; в обоих случаях используется 16 битов:

десятичное: 0001 1001 0100 0100 двоичное: 0000011110011000

16 битов в двоично-десятичном формате могут хранить числа от 0 до 9999, то есть всего 10000 различных комбинаций, а 16 битов в двоичном формате — 65536 комбинаций. Именно по этой причине говорят, что двоичная система эффективнее.

Однако представим, что могло бы произойти, если бы какой-нибудь гениальный молодой инженер придумал очень надежное электронное устройство, которое могло бы хранить разряды от 0 до 9, разделив участок напряжения от 0 до 10 В на 10 интервалов. Четыре таких устройства могли бы хранить десятичное число от 0 до 9999, то есть 10 000 комбинаций. А если бы те же устройства использовались для хранения двоичных чисел, они могли бы содержать всего 16 комбинаций. Естественно, в этом случае десятичная система была бы более эффективной.

Адреса памяти

Память состоит из **ячеек**, каждая из которых может хранить некоторую порцию информации. Каждая ячейка имеет номер, который называется **адресом**. По адресу программы могут ссылаться на определенную ячейку. Если память содержит n ячеек, они будут иметь адреса от 0 до n–1. Все ячейки памяти содержат одинаковое число битов. Если ячейка состоит из k битов, она может содержать любую из 2^k комбинаций. На рис. 2.8 показаны 3 различных способа организации 96-битной памяти. Отметим, что соседние ячейки по определению имеют последовательные адреса.

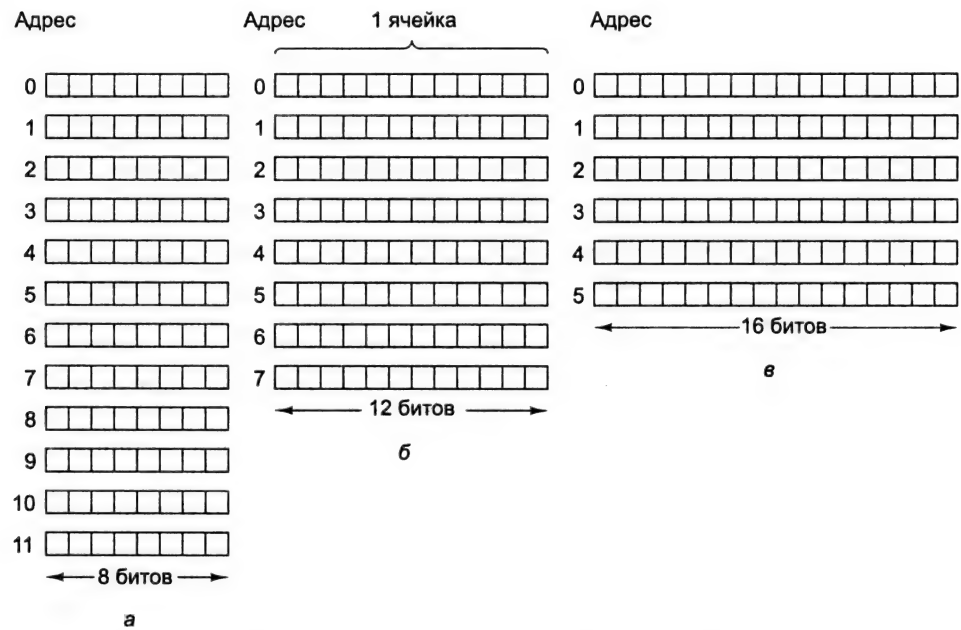


Рис. 2.8. Три способа организации 96-битной памяти

В компьютерах, где используется двоичная система счисления (включая восьмеричное и шестнадцатеричное представление двоичных чисел), адреса памяти также выражаются в двоичных числах. Если адрес состоит из m битов, максимальное число адресованных ячеек будет составлять 2^m. Например, адрес для обращения к памяти, изображенной на рис. 2.8, *а*, должен состоять по крайней мере из

4 битов, чтобы выражать все числа от 0 до 11. При устройстве памяти, показанном на рис. 2.8, *б* и 2.8, *в*, достаточно 3-битного адреса. Число битов в адресе определяет максимальное количество адресованных ячеек памяти и не зависит от числа битов в ячейке. 12-битные адреса нужны и памяти с 2^{12} ячеек по 8 битов каждая, и памяти с 2^{12} ячеек по 64 бита каждая.

В табл. 2.1 показано число битов в ячейке для некоторых коммерческих компьютеров.

Таблица 2.1. Число битов в ячейке для некоторых моделей коммерческих компьютеров

Компьютер	Число битов в ячейке
Burroughs B1700	1
IBM PC	8
DEC PDP-8	12
IBM 1130	16
DEC PDP-15	18
XDS 940	24
Electrologica X8	27
XDS Sigma 9	32
Honeywell 6180	36
CDC 3600	48
CDC Syber	60

Ячейка — минимальная единица, к которой можно обращаться. В последние годы практически все производители выпускают компьютеры с 8-битными ячейками, которые называются **байтами**. Байты группируются в **слова**. Компьютер с 32-битными словами имеет 4 байта на каждое слово, а компьютер с 64-битными словами — 8 байтов на каждое слово. Такая единица, как слово, необходима, поскольку большинство команд производят операции над целыми словами (например, складывают два слова). Таким образом, 32-битная машина будет содержать 32-битные регистры и команды для манипуляций с 32-битными словами, тогда как 64-битная машина будет иметь 64-битные регистры и команды для перемещения, сложения, вычитания и других операций над 64-битными словами.

Упорядочение байтов

Байты в слове могут нумероваться слева направо или справа налево. На первый взгляд может показаться, что между этими двумя вариантами нет разницы, но мы скоро увидим, что выбор имеет большое значение. На рис. 2.9, *а* изображена часть памяти 32-битного компьютера, в котором байты пронумерованы слева направо (как у компьютеров SPARC или больших IBM). Рисунок 2.9, *б* показывает аналогичную репрезентацию 32-битного компьютера с нумерацией байтов справа налево (как у компьютеров Intel).

Важно понимать, что в обеих системах 32-битное целое число (например, 6) представлено битами 110 в трех крайних правых битах слова, а остальные 29 битов

представлены нулями. Если байты нумеруются слева направо, биты 110 находятся в байте 3 (или 7, или 11 и т. д.). Если байты нумеруются справа налево, биты 110 находятся в байте 0 (или 4, или 8 и т. д.). В обоих случаях слово, содержащее это целое число, имеет адрес 0.

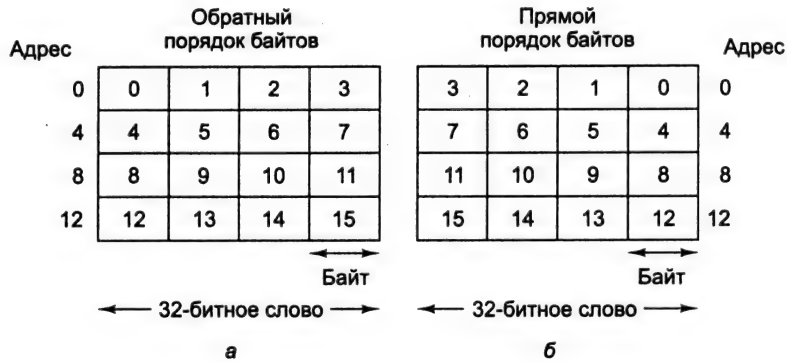


Рис. 2.9. Память с нумерацией байтов слева направо (*а*); память с нумерацией байтов справа налево (*б*)

Если компьютеры содержат только целые числа, никаких сложностей не возникает. Однако многие прикладные задачи требуют использования не только целых чисел, но и цепочек символов и других типов данных. Рассмотрим, например, простую запись данных персонала, состоящую из цепочки символов (имя сотрудника) и двух целых чисел (возраст и номер отдела). Цепочка символов завершается одним или несколькими байтами 0, чтобы заполнить слово. На рис. 2.10, *а* представлена схема с нумерацией байтов слева направо, а на рис. 2.10, *б* — с нумерацией байтов справа налево для записи «Jim Smith, 21 год, отдел 260» (1×256+4=260).



Рис. 2.10. Запись данных сотрудника для машины с нумерацией байтов слева направо (*а*); та же запись для машины с нумерацией байтов справа налево (*б*); результат преобразования записи из нумерации слева направо в нумерацию справа налево (*в*); результат перестановки байтов в предыдущем случае (*г*)

Оба эти представления хороши и внутренне последовательны. Проблемы начинаются тогда, когда один из компьютеров пытается переслать эту запись на другой компьютер по сети. Предположим, что машина с нумерацией байтов слева на-

право пересылает запись на компьютер с нумерацией байтов справа налево по одному байту, начиная с байта 0 и заканчивая байтом 19. Для простоты будем считать, что биты не инвертируются при передаче. Таким образом, байт 0 переносится из первой машины на вторую в байт 0 и т. д., как показано на рис. 2.10, *в*.

Компьютер, получивший запись, имя печатает правильно, но возраст получается 21×2^{24}, и номер отдела тоже искажается. Такая ситуация возникает, поскольку при передаче записи порядок букв в слове меняется так, как нужно, но при этом порядок байтов целых чисел тоже изменяется, что приводит к неверному результату.

Очевидное решение этой проблемы — наличие программного обеспечения, которое инвертировало бы байты в слове после того, как сделана копия. Результат такой операции изображен на рис. 2.10, *г*. Мы видим, что числа стали правильными, но цепочка символов превратилась в «MIJTIMS», при этом «H» вообще поместилась отдельно. Цепочка переворачивается потому, что компьютер сначала считывает байт 0 (пробел), затем байт 1 (M) и т. д.

Простого решения не существует. Есть один способ, но он неэффективен. (Нужно перед каждой единицей данных помещать заголовок, информирующий, какой тип данных последует за ним — цепочка, целое число и т. д. Это позволит компьютеру-получателю производить только необходимые преобразования.) Ясно, что отсутствие стандарта упорядочивания байтов является главным неудобством при обмене информацией между разными машинами.

Код с исправлением ошибок

Память компьютера время от времени может делать ошибки из-за всплесков напряжения на линии электропередачи и по другим причинам. Чтобы бороться с такими ошибками, используются коды с обнаружением и исправлением ошибок. При этом к каждому слову в памяти особым образом добавляются дополнительные биты. Когда слово считывается из памяти, эти биты проверяются на наличие ошибок.

Чтобы понять, как обращаться с ошибками, необходимо внимательно изучить, что представляют собой эти ошибки. Предположим, что слово состоит из m битов данных, к которым мы прибавляем r дополнительных битов (контрольных разрядов). Пусть общая длина слова будет n (то есть $n=m+r$). n-битную единицу, содержащую m битов данных и r контрольных разрядов, часто называют **кодированным словом**.

Для любых двух кодированных слов, например 10001001 и 10110001, можно определить, сколько соответствующих битов в них различается. В данном примере таких бита три. Чтобы определить количество различающихся битов, нужно над двумя кодированными словами произвести логическую операцию ИСКЛЮЧАЮЩЕЕ ИЛИ и сосчитать число битов со значением 1 в полученном результате. Число битовых позиций, по которым различаются два слова, называется **интервалом Хэмминга**. Если интервал Хэмминга для двух слов равен d, это значит, что достаточно d битовых ошибок, чтобы превратить одно слово в другое. Например, интервал Хэмминга кодированных слов 11110001 и 00110000 равен 3, поскольку для превращения первого слова во второе достаточно 3 ошибок в битах.

Память состоит из m-битных слов, и следовательно, существует 2^m вариантов сочетания битов. Кодированные слова состоят из n битов, но из-за способа под-

счета контрольных разрядов допустимы только 2^m из 2^n кодированных слов. Если в памяти обнаруживается недопустимое кодированное слово, компьютер знает, что произошла ошибка. При наличии алгоритма для подсчета контрольных разрядов можно составить полный список допустимых кодированных слов и из этого списка найти два слова, для которых интервал Хэмминга будет минимальным. Это интервал Хэмминга полного кода.

Свойства проверки и исправления ошибок определенного кода зависят от его интервала Хэмминга. Чтобы обнаружить d ошибок в битах, необходим код с интервалом d+1, поскольку d ошибок не могут изменить одно допустимое кодированное слово на другое допустимое кодированное слово. Соответственно, чтобы исправить d ошибок в битах, необходим код с интервалом 2d+1, поскольку в этом случае допустимые кодированные слова так сильно отличаются друг от друга, что даже если произойдет d изменений, изначальное кодированное слово будет ближе к ошибочному, чем любое другое кодированное слово, поэтому его без труда можно будет определить.

В качестве простого примера кода с обнаружением ошибок рассмотрим код, в котором к данным присоединяется один **бит четности**. Бит четности выбирается таким образом, что число битов со значением 1 в кодированном слове четное (или нечетное). Интервал этого кода равен 2, поскольку любая ошибка в битах приводит к кодированному слову с неправильной четностью. Другими словами, достаточно двух ошибок в битах для перехода от одного допустимого кодированного слова к другому допустимому слову. Такой код может использоваться для обнаружения одиночных ошибок. Если из памяти считывается слово, содержащее неверную четность, поступает сигнал об ошибке. Программа не сможет продолжаться, но зато не будет неверных результатов.

В качестве простого примера кода с исправлением ошибок рассмотрим код с четырьмя допустимыми кодированными словами:

0000000000, 0000011111, 1111100000 и 1111111111

Интервал этого кода равен 5. Это значит, что он может исправлять двойные ошибки. Если появляется кодированное слово 0000000111, компьютер знает, что изначальное слово должно быть 0000011111 (если произошло не более двух ошибок). При наличии трех ошибок, если, например, слово 0000000000 изменилось на 0000000111, этот метод недопустим.

Представим, что мы хотим разработать код с m битами данных и r контрольных разрядов, который позволил бы исправлять все ошибки в битах. Каждое из 2^m допустимых слов имеет n недопустимых кодированных слов, которые отличаются от допустимого одним битом. Они образуются инвертированием каждого из n битов в n-битном кодированном слове. Следовательно, каждое из 2^m допустимых слов требует n+1 возможных сочетаний битов, приписываемых этому слову (n возможных ошибочных вариантов и один правильный). Поскольку общее число различных сочетаний битов равно 2^n, то $(n+1)2^m \leq 2^n$. Так как $n=m+r$, следовательно, $(m+r+1) \leq 2^r$. Эта формула дает нижний предел числа контрольных разрядов, необходимых для исправления одиночных ошибок. В табл. 2.2 показано необходимое количество контрольных разрядов для слов разного размера.

Таблица 2.2. Число контрольных разрядов для кода, способного исправлять одиночные ошибки

Размер слова	Количество контрольных разрядов	Общий размер	На сколько процентов увеличилась длина слова
8	4	12	50
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	265	4
512	10	522	2

Этого теоретического нижнего предела можно достичь, используя метод Ричарда Хэмминга. Но прежде чем обратиться к этому алгоритму, давайте рассмотрим простую графическую схему, которая четко иллюстрирует идею кода с исправлением ошибок для 4-битных слов. Диаграмма Венна на рис. 2.11 содержит 3 круга, A, B и C, которые вместе образуют семь секторов. Давайте закодируем в качестве примера слово из 4 битов 1100 в сектора AB, ABC, AC и BC, по одному биту в каждом секторе (в алфавитном порядке). Кодирование показано на рис. 2.11, *а*.

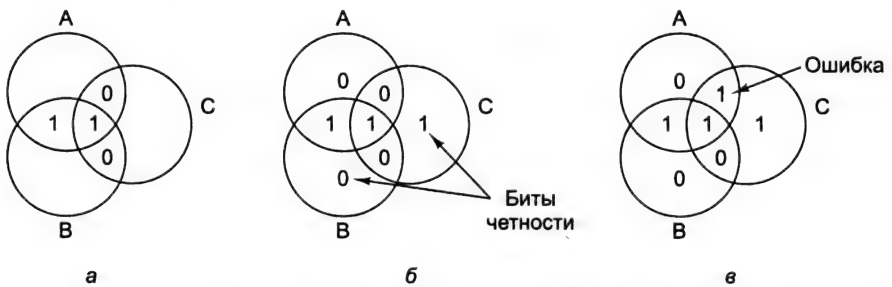


Рис. 2.11. Кодировка числа 1100 (*а*); добавляются биты четности (*б*); ошибка в секторе AC (*в*)

Далее мы добавим бит четности к каждому из трех пустых секторов, чтобы получилась положительная четность, как показано на рис. 2.11, *б*. По определению сумма битов в каждом из трех кругов, A, B, и C, должна быть четной. В круге A находится 4 числа: 0, 0, 1 и 1, которые в сумме дают четное число 2. В круге B находятся числа 1, 1, 0 и 0, которые также при сложении дают четное число 2. То же имеет силу и для круга C. В данном примере получилось так, что все суммы одинаковы, но вообще возможны случаи с суммами 0 и 4. Рисунок соответствует кодированному слову, состоящему из 4 битов данных и 3 битов четности.

Предположим, что бит в секторе AC изменился с 0 на 1, как показано на рис. 2.11, *в*. Компьютер видит, что круги A и C имеют отрицательную четность. Единственный способ исправить ошибку, изменив только один бит, — возвращение биту AC значения 0. Таким способом компьютер может исправлять одиночные ошибки автоматически.

А теперь посмотрим, как может использоваться алгоритм Хэмминга при создании кодов с исправлением ошибок для слов любого размера. В коде Хэмминга к

слову, состоящему из m битов, добавляется r битов четности, при этом образуется слово длиной m+r битов. Биты нумеруются с единицы (а не с нуля), причем первым считается крайний левый. Все биты, номера которых — степени двойки, являются битами четности; остальные используются для данных. Например, к 16-битному слову нужно добавить 5 битов четности. Биты с номерами 1, 2, 4, 8 и 16 — биты четности, а все остальные — биты данных. Всего слово содержит 21 бит (16 битов данных и 5 битов четности). В рассматриваемом примере мы будем использовать положительную четность (выбор произвольный).

Каждый бит четности проверяет определенные битовые позиции. Общее число битов со значением 1 в проверяемых позициях должно быть четным. Ниже указаны позиции проверки для каждого бита четности:

Бит 1 проверяет биты 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21.
Бит 2 проверяет биты 2, 3, 6, 7, 10, 11, 14, 15, 18, 19.
Бит 4 проверяет биты 4, 5, 6, 7, 12, 13, 14, 15, 20, 21.
Бит 8 проверяет биты 8, 9, 10, 11, 12, 13, 14, 15.
Бит 16 проверяет биты 16, 17, 18, 19, 20, 21.

В общем случае бит b проверяется битами $b_1, b_2, \ldots, b_j$, такими что $b_1+b_2+\ldots+b_j=b$. Например, бит 5 проверяется битами 1 и 4, поскольку 1+4=5. Бит 6 проверяется битами 2 и 4, поскольку 2+4=6 и т. д.

На рис. 2.12 показано построение кода Хэмминга для 16-битного слова 1111000010101110. Соответствующим 21-битным кодированным словом является 001011100000101101110. Чтобы увидеть, как происходит исправление ошибок, рассмотрим, что произойдет, если бит 5 изменит значение из-за резкого скачка напряжения на линии электропередачи. В результате вместо кодированного слова 001011100000101101110 получится 001001100000101101110. Будут проверены 5 битов четности. Вот результаты проверки:

Бит четности 1 неправильный (биты 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21 содержат пять единиц).

Бит четности 2 правильный (биты 2, 3, 6, 7, 10, 11, 14, 15, 18, 19 содержат шесть единиц).

Бит четности 4 неправильный (биты 4, 5, 6, 7, 12, 13, 14, 15, 20, 21 содержат пять единиц).

Бит четности 8 правильный (биты 8, 9, 10, 11, 12, 13, 14, 15 содержат две единицы).

Бит четности 16 правильный (биты 16, 17, 18, 19, 20, 21 содержат четыре единицы).

Общее число единиц в битах 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 и 21 должно быть четным, поскольку в данном случае используется положительная четность. Неправильным должен быть один из битов, проверяемых битом четности 1 (а именно 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 и 21). Бит четности 4 тоже неправильный. Это значит, что изменил значение один из следующих битов: 4, 5, 6, 7, 12, 13, 14, 15, 20, 21. Ошибка должна быть в бите, который содержится в обоих списках. В данном случае общими являются биты 5, 7, 13, 15 и 21. Поскольку бит четности 2 правильный, биты 7 и 15 исключаются. Правильность бита четности 8 исключает наличие ошибки в бите 13. Наконец, бит 21 также исключается, поскольку бит четности 16 правильный. В итоге остается бит 5, в котором и содержится ошибка. Поскольку этот бит имеет значение 1, он должен принять значение 0. Именно таким образом исправляются ошибки.

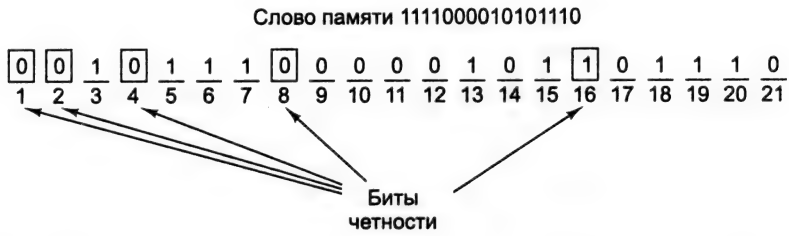


Рис. 2.12. Построение кода Хэмминга для слова 1111000010101110 с помощью добавления 5 контрольных разрядов к 16 битам данных

Чтобы найти неправильный бит, сначала нужно подсчитать все биты четности. Если они правильные, ошибки нет (или есть, но больше одной). Если обнаружились неправильные биты четности, то нужно сложить их номера. Сумма, полученная в результате, даст номер позиции неправильного бита. Например, если биты четности 1 и 4 неправильные, а 2, 8 и 16 правильные, то ошибка произошла в бите 5 (1+4).

Кэш-память

Процессоры всегда работали быстрее, чем память. Процессоры и память совершенствовались параллельно, поэтому это несоответствие сохранялось. Поскольку на микросхему можно помещать все больше и больше транзисторов, разработчики процессоров использовали эти преимущества для создания конвейеров и суперскалярной архитектуры, что еще больше повышало скорость работы процессоров. Разработчики памяти обычно использовали новые технологии для увеличения емкости, а не скорости, что еще больше усугубляло проблему. На практике такое несоответствие в скорости работы приводит к следующему: после того как процессор дает запрос памяти, должно пройти много циклов, прежде чем он получит слово, которое ему нужно. Чем медленнее работает память, тем дольше процессору приходится ждать, тем больше циклов должно пройти.

Как мы уже говорили выше, есть два пути решения этой проблемы. Самый простой из них — начать считывать информацию из памяти, когда это необходимо, и при этом продолжать выполнение команд, но если какая-либо команда попытается использовать слово до того, как оно считалось из памяти, процессор должен приостанавливать работу. Чем медленнее работает память, тем чаще будет возникать такая проблема и тем больше будет проигрыш в работе. Например, если отсрочка составляет 10 циклов, весьма вероятно, что одна из 10 следующих команд попытается использовать слово, которое еще не считалось из памяти.

Другое решение проблемы — сконструировать машину, которая не приостанавливает работу, но следит, чтобы программы-компиляторы не использовали слова до того, как они считаются из памяти. Однако это не так просто осуществить на практике. Часто при выполнении команды загрузки машина не может выполнять другие действия, поэтому компилятор вынужден вставлять пустые команды, которые не производят никаких операций, но при этом занимают место в памяти. В действительности при таком подходе простаивает не аппаратное, а программное обеспечение, но снижение производительности при этом такое же.

На самом деле эта проблема не технологическая, а экономическая. Инженеры знают, как построить память, которая будет работать так же быстро, как и процессор, но при этом ее приходится помещать прямо на микросхему процессора (поскольку информация через шину поступает очень медленно). Установка большой памяти на микросхему процессора делает его больше и, следовательно, дороже, и даже если бы стоимость не имела значения, все равно существуют ограничения в размерах процессора, который можно сконструировать. Таким образом, приходится выбирать между быстрой памятью небольшого размера и медленной памятью большого размера. Мы бы предпочли память большого размера с высокой скоростью работы по низкой цене.

Интересно отметить, что существуют технологии сочетания маленькой и быстрой памяти с большой и медленной, что позволяет получить и высокую скорость работы, и большую емкость по разумной цене. Маленькая память с высокой скоростью работы называется **кэш-памятью** (от французского слова cacher «прятать»[1]; читается «кэш»). Ниже мы кратко опишем, как используется кэш-память и как она работает. Более подробное описание см. в главе 4.

Основная идея кэш-памяти проста: в ней находятся слова, которые чаще всего используются. Если процессору нужно какое-нибудь слово, сначала он обращается к кэш-памяти. Только в том случае, если слова там нет, он обращается к основной памяти. Если значительная часть слов находится в кэш-памяти, среднее время доступа значительно сокращается.

Таким образом, успех или неудача зависит от того, какая часть слов находится в кэш-памяти. Давно известно, что программы не обращаются к памяти наугад. Если программе нужен доступ к адресу A, то скорее всего после этого ей понадобится доступ к адресу, расположенному поблизости от A. Практически все команды обычной программы (за исключением команд перехода и вызова процедур) вызываются из последовательных участков памяти. Кроме того, большую часть времени программа тратит на циклы, когда ограниченный набор команд выполняется снова и снова. Точно так же при манипулировании матрицами программа скорее всего будет обращаться много раз к одной и той же матрице, прежде чем перейдет к чему-либо другому.

То, что при последовательных отсылках к памяти в течение некоторого промежутка времени используется только небольшой ее участок, называется **принципом локальности**. Этот принцип составляет основу всех систем кэш-памяти. Идея состоит в следующем: когда определенное слово вызывается из памяти, оно вместе с соседними словами переносится в кэш-память, что позволяет при очередном запросе быстро обращаться к следующим словам. Общее устройство процессора, кэш-памяти и основной памяти показано на рис. 2.13. Если слово считывается или записывается k раз, компьютеру понадобится сделать 1 обращение к медленной основной памяти и k–1 обращений к быстрой кэш-памяти. Чем больше k, тем выше общая производительность.

---

[1] В английском «cash» получило значение «наличные (карманные) деньги», то есть то, что под рукой. А уже из него и образовался термин «кэш», который относят к сверхоперативной памяти. — *Примеч. научн. ред.*

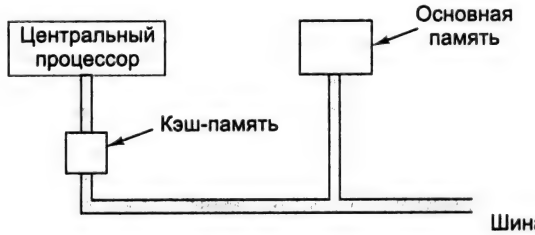


Рис. 2.13. Кэш-память по логике вещей должна находиться между процессором и основной памятью. В действительности существует три возможных варианта расположения кэш-памяти

Мы можем сделать более строгие вычисления. Пусть c — время доступа к кэш-памяти, m — время доступа к основной памяти и h — **коэффициент совпадения**, который показывает соотношение числа ссылок к кэш-памяти и общего числа всех ссылок. В нашем примере h=(k–1)/k. Таким образом, мы можем вычислить среднее время доступа:

среднее время доступа =c+(1–h)m.

Если h→1 и все обращения делаются только к кэш-памяти, то время доступа стремится к c. С другой стороны, если h→0 и каждый раз нужно обращаться к основной памяти, то время доступа стремится к c+m: сначала требуется время c для проверки кэш-памяти (в данном случае безуспешной), а затем время m для обращения к основной памяти. В некоторых системах обращение к основной памяти может начинаться параллельно с исследованием кэш-памяти, чтобы в случае неудачного поиска цикл обращения к основной памяти уже начался. Однако эта стратегия требует способности останавливать процесс обращения к основной памяти в случае результативного обращения к кэш-памяти, что делает разработку такого компьютера более сложной.

Основная память и кэш-память делятся на блоки фиксированного размера с учетом принципа локальности. Блоки внутри кэш-памяти обычно называют **строками кэш-памяти (cache lines)**. Если обращение к кэш-памяти нерезультативно, из основной памяти в кэш-память загружается вся строка, а не только необходимое слово. Например, если строка состоит из 64 байтов, обращение к адресу 260 повлечет за собой загрузку в кэш-память всей строки, то есть с 256-го по 319-й байт. Возможно, через некоторое время понадобятся другие слова из этой строки. Такой путь обращения к памяти более эффективен, чем вызов каждого слова по отдельности, потому что вызвать k слов 1 раз можно гораздо быстрее, чем 1 слово k раз. Если входные сообщения кэш-памяти содержат более одного слова, это значит, что будет меньше таких входных сообщений и, следовательно, меньше непроизводительных затрат.

Разработка кэш-памяти очень важна для процессоров с высокой производительностью. Первый вопрос — размер кэш-памяти. Чем больше размер, тем лучше работает память, но тем дороже она стоит. Второй вопрос — размер строки кэш-памяти. Кэш-память объемом 16 Кбайт можно разделить на 1K строк по 16 байтов, 2K строк по 8 байтов и т. д. Третий вопрос — как устроена кэш-память, то есть как она определяет, какие именно слова содержатся в ней в данный момент. Устройство кэш-памяти мы рассмотрим подробно в главе 4.

Четвертый вопрос — должны ли команды и данные находиться вместе в общей кэш-памяти. Проще разработать **смежную кэш-память**, в которой хранятся и данные, и команды. При этом вызов команд и данных автоматически уравновешивается. Тем не менее в настоящее время существует тенденция к использованию **разделенной кэш-памяти**, когда команды хранятся в одной кэш-памяти, а данные — в другой. Такая структура также называется **Гарвардской (Harvard Architecture)**, поскольку идея использования отдельной памяти для команд и отдельной памяти для данных впервые воплотилась в компьютере Марс III, который был создан Говардом Айкеном в Гарварде. Современные разработчики пошли по этому пути, поскольку сейчас широко используются процессоры с конвейерами, а при такой организации должна быть возможность одновременного доступа и к командам, и к данным (операндам). Разделенная кэш-память позволяет осуществлять параллельный доступ, а общая — нет. К тому же, поскольку команды обычно не меняются во время выполнения, содержание командной кэш-памяти никогда не приходится записывать обратно в основную память.

Наконец, пятый вопрос — количество блоков кэш-памяти. В настоящее время очень часто кэш-память первого уровня располагается прямо на микросхеме процессора, кэш-память второго уровня — не на самой микросхеме, но в корпусе процессора, а кэш-память третьего уровня — еще дальше от процессора.

Сборка модулей памяти и их типы

Со времен появления полупроводниковой памяти и до начала 90-х годов все микросхемы памяти производились, продавались и устанавливались на плату компьютера по отдельности. Эти микросхемы вмещали от 1 Кбит до 1 Мбит информации и выше. В первых персональных компьютерах часто оставлялись пустые разъемы, чтобы покупатель в случае необходимости мог вставить дополнительные микросхемы.

В настоящее время распространен другой подход. Группа микросхем (обычно 8 или 16) монтируется на одну крошечную печатную плату и продается как один блок. Он называется **SIMM (Single Inline Memory Module — модуль памяти, имеющий выводы с одной стороны)** или **DIMM (Dual Inline Memory Module — модуль памяти, у которого выводы расположены с двух сторон)**. У первого из них контакты расположены только на одной стороне печатной платы (выводы на второй стороне дублируют первую), а у второго — на обеих сторонах. Схема SIMM изображена на рис. 2.14.

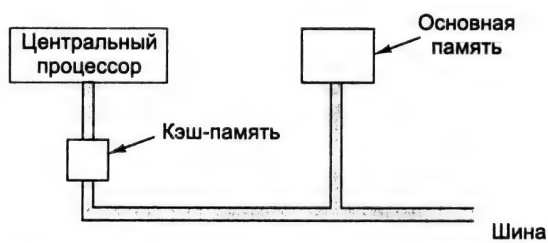


Рис. 2.14. Модуль SIMM в 32 Мбайт. Модулем управляют две микросхемы

Обычный модуль SIMM содержит 8 микросхем по 32 Мбит (4 Мбайт) каждая. Таким образом, весь модуль вмещает 32 Мбайт информации. Во многие компьютеры встраивается 4 модуля, следовательно, при использовании модулей SIMM по 32 Мбайт общий объем памяти составляет 128 Мбайт. При необходимости данные модули SIMM можно заменить модулями с большей вместимостью (64 Мбайт и выше).

У первых модулей SIMM было 30 контактов, и они могли передавать 8 битов информации за один раз. Остальные контакты использовались для адресации и контроля. Более поздние модули содержали уже 72 контакта и передавали 32 бита информации за один раз. Для компьютера Pentium, который требовал одновременной передачи 64 битов, эти модули соединялись по два, и каждый из них доставлял половину требуемых битов. В настоящее время стандартным способом сборки является модуль DIMM. У него на каждой стороне платы находится по 84 позолоченных контакта, то есть всего 168. DIMM способен передавать 64 бита данных за раз. Вместимость DIMM обычно составляет 64 Мбайт и выше. В электронных записных книжках обычно используется модуль DIMM меньшего размера, который называется **SO-DIMM (Small Outline DIMM)**. Модули SIMM и DIMM могут содержать бит четности или код исправления ошибок, однако, поскольку вероятность возникновения ошибок в модуле 1 ошибка в 10 лет, в большинстве обычных компьютеров методы обнаружения и исправления ошибок не применяются.

Вспомогательная память

Не важно, каков объем основной памяти: он все равно всегда будет слишком мал. Мы всегда хотим хранить в памяти компьютера больше информации, чем она может вместить. С развитием технологий людям приходят в голову такие вещи, которые раньше считались совершенно фантастическими. Например, можно вообразить, что Библиотека Конгресса решила представить в цифровой форме и продать полное содержание всех хранящихся в ней изданий в одной статье («Все человеческие знания всего за $49»). В среднем каждая книга содержит 1 Мбайт текста и 1 Мбайт сжатых рисунков. Таким образом, для размещения 50 млн книг понадобится 10^{14} байт или 100 Тбайт памяти. Для хранения всех существующих художественных фильмов (50 000) необходимо примерно столько же места. Такое количество информации в настоящее время невозможно разместить в основной памяти, и вряд ли можно будет это сделать в будущем (по крайней мере, в ближайшие несколько десятилетий).

Иерархическая структура памяти

Иерархическая структура памяти является традиционным решением проблемы хранения большого количества данных. Она изображена на рис. 2.15. На самом верху находятся регистры процессора. Доступ к регистрам осуществляется быстрее всего. Дальше идет кэш-память, объем которой сейчас составляет от 32 Кбайт до нескольких мегабайт. Затем следует основная память, которая в настоящее

время может вмещать от 16 Мбайт до десятков гигабайтов. Далее идут магнитные диски и, наконец, накопители на магнитной ленте и оптические диски, которые используются для хранения архивной информации.



Рис. 2.15. Пятиуровневая организация памяти

По мере продвижения по структуре сверху вниз возрастают три параметра. Во-первых, увеличивается время доступа. Доступ к регистрам занимает несколько наносекунд, доступ к кэш-памяти — немного больше, доступ к основной памяти — несколько десятков наносекунд. Дальше идет большой разрыв: доступ к дискам занимает по крайней мере 10 мкс, а время доступа к магнитным лентам и оптическим дискам вообще может измеряться в секундах (поскольку эти накопители информации еще нужно взять и поместить в соответствующее устройство).

Во-вторых, увеличивается объем памяти. Регистры могут содержать в лучшем случае 128 байтов, кэш-память — несколько мегабайтов, основная память — десятки тысяч мегабайтов, магнитные диски — от нескольких гигабайтов до нескольких десятков гигабайтов. Магнитные ленты и оптические диски хранятся автономно от компьютера, поэтому их объем ограничивается только финансовыми возможностями владельца.

В-третьих, увеличивается количество битов, которое вы получаете за 1 доллар. Стоимость объема основной памяти измеряется в долларах за мегабайт[1], объем магнитных дисков — в пенни за мегабайт, а объем магнитной ленты — в долларах за гигабайт или еще дешевле.

Регистры, кэш-память и основную память мы уже рассмотрели. В следующих разделах мы расскажем о магнитных дисках, а затем приступим к изучению оптических дисков. Накопители на магнитных лентах мы рассматривать не будем, поскольку они очень редко используются; к тому же о них практически нечего сказать.

---

[1] Заметим, что стоимость памяти постоянно уменьшается, в то время как ее объем — увеличивается. Закон Мура применим и здесь. Сегодня 1 Мбайт оперативной памяти стоит около 10 центов. — *Примеч. научн. ред.*

Магнитные диски

Магнитный диск состоит из одного или нескольких алюминиевых дисков[1] с магнитным слоем. Изначально они были 50 см в диаметре, но сейчас их диаметр составляет от 3 до 12 см, а у портативных компьютеров — меньше 3 см, причем этот параметр продолжает уменьшаться. Головка диска, содержащая индукционную катушку, двигается над поверхностью диска, опираясь на воздушную подушку. Отметим, что у дискет головка касается поверхности. Когда через головку проходит положительный или отрицательный ток, он намагничивает поверхность под головкой. При этом магнитные частицы намагничиваются направо или налево в зависимости от полярности тока. Когда головка проходит над намагниченной областью, в ней (в головке) возникает положительный или отрицательный ток, что дает возможность считывать записанные ранее биты. Поскольку диск вращается под головкой, поток битов может записываться, а потом считываться. Конфигурация дорожки диска показана на рис. 2.16.

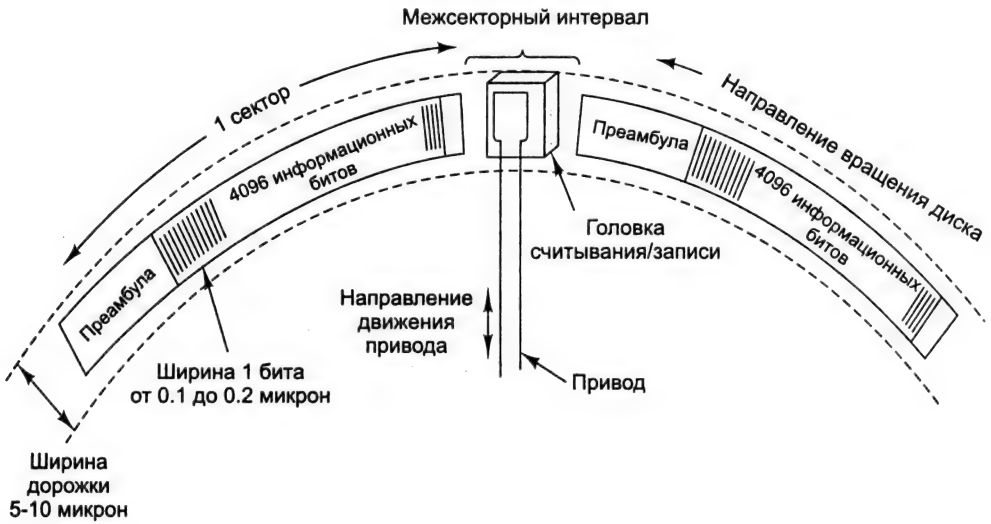


Рис. 2.16. Кусок дорожки диска (два сектора)

Дорожкой называется круговая последовательность битов, записанных на диск за его полный оборот. Каждая дорожка делится на **секторы** фиксированной длины. Каждый сектор обычно содержит 512 байтов данных. Перед данными располагается **преамбула (preamble)**, которая позволяет головке синхронизироваться перед чтением или записью. После данных идет код с исправлением ошибок (код Хэмминга или чаще **код Рида—Соломона**, который может исправлять много ошибок, а не только одиночные). Между соседними секторами находится **межсекторный интервал**. Многие производители указывают размер неформатированного диска (как будто каждая дорожка содержит только данные), но более честно было бы указывать вместимость форматированного диска, когда не учитываются пре-

[1] В настоящее время фирма IBM делает их из стекла. — *Примеч. научн. ред.*

амбулы, коды с исправлением ошибок и межсекторные интервалы. Емкость форматированного диска обычно на 15% меньше емкости неформатированного диска.

У всех дисков есть кронштейны, они могут перемещаться туда и обратно по радиусу на разные расстояния от шпинделя, вокруг которого вращается диск. На разных расстояниях от оси записываются разные дорожки. Таким образом, дорожки представляют собой ряд концентрических кругов, расположенных вокруг шпинделя. Ширина дорожки зависит от величины головки и от точности ее перемещения. На сегодняшний момент диски имеют от 800 до 2000 дорожек на см[1], то есть ширина каждой дорожки составляет от 5 до 10 микрон (1микрон=1/1000 мм). Следует отметить, что дорожка — это не углубление на поверхности диска, а просто кольцо намагниченного материала, которое отделяется от других дорожек небольшими пограничными областями.

Плотность записи битов на концентрических дорожках различная, в зависимости от расстояния от центра диска. Плотность записи зависит главным образом от качества поверхности диска и чистоты воздуха. Плотность записи современных дисков разнится от 50 000 до 100 000 бит/см. Чтобы достичь высокого качества поверхности и достаточной чистоты воздуха, диски герметично запечатываются, что защищает их от попадания грязи. Такие диски называются **винчестерами**. Впервые они были выпущены фирмой IBM. У них было 30 Мбайт фиксированной памяти и 30 Мбайт сменной памяти. Возможно, эти диски 30–30 ассоциировались с ружьями «Винчестер» 30–30[2]. Большинство магнитных дисков состоит из нескольких пластин, расположенных друг под другом, как показано на рис. 2.17. Каждая поверхность снабжена рычагом и головкой. Рычаги скреплены таким образом, что одновременно могут перемещаться на разные расстояния от оси. Совокупность дорожек, расположенных на одном расстоянии от центра, называется **цилиндром**.

Производительность диска зависит от многих факторов. Чтобы считать или записать сектор, головка должна переместиться на нужное расстояние от оси. Этот процесс называется **поиском**. Среднее время поиска между дорожками, взятыми наугад, составляет от 5 до 15 мс, а поиск между последовательными дорожками занимает около 1 мс. Когда головка помещается на нужное расстояние от центра, выжидается некоторое количество времени (оно называется **временем ожидания сектора**), пока нужный сектор не оказывается под головкой. Большинство дисков вращаются со скоростью 3600, 5400 или 7200 оборотов в минуту. Таким образом, среднее время ожидания сектора (половина оборота) составляет от 4 до 8 мс. Существуют также диски со скоростью вращения 10800 оборотов в минуту (180 оборотов в секунду). Время передачи информации зависит от плотности записи и скорости вращения. При скорости передачи от 5 до 10 Мбайт в секунду[3] время передачи одного сектора (512 байтов) составляет от 25 до 100 мкс. Следовательно, время поиска и время ожидания сектора определяет время передачи информации. Ясно, что считывание секторов из разных частей диска неэффективно.

---

[1] Плотность хранения информации на магнитных дисках также постоянно увеличивается, и сейчас на 1 см поверхности уже размещается более 10000 дорожек. — *Примеч. научн. ред.*

[2] Двуствольное ружье 30-го калибра. — *Примеч. перев.*

[3] В современных винчестерах скорость линейного чтения уже превысила 40 Мбайт в секунду. — *Примеч. научн. ред.*

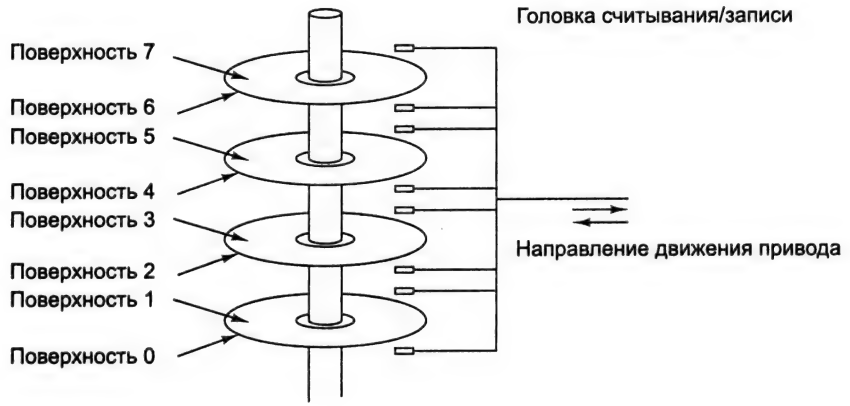


Рис. 2.17. Винчестер с четырьмя дисками

Следует упомянуть, что из-за наличия преамбул, кодов с исправлением ошибок, промежутков между секторами, а также из-за того, что определенное время затрачивается на поиск дорожки и на ожидание сектора, существует огромная разница между максимальной скоростью передачи данных, когда необходимые данные разбросаны в разных частях диска, и той, когда они находятся в одном месте и считываются последовательно. Максимальная скорость передачи данных в первом случае достигается в тот момент, когда головка расположена над первым битом данных. Однако такая скорость работы может сохраняться только на одном секторе. Для некоторых приложений, например мультимедиа, имеет значение именно средняя скорость передачи за некоторый период с учетом необходимого времени поиска и времени ожидания сектора.

Поскольку диски вращаются со скоростью от 60 до 120 оборотов в секунду, они нагреваются и расширяются, то есть физически изменяются в размерах. Некоторые диски должны периодически совершать рекалибровку механизмов перемещения, чтобы компенсировать эти расширения. Поэтому мощность привода, перемещающего головки над поверхностью диска, периодически меняется. При таких рекалибровках могут возникать трудности с использованием прикладных программ мультимедиа, которые ожидают более или менее непрерывного потока битов, поступающего с максимальной скоростью передачи последовательной информации (с одной части диска). Для работы с прикладными программами мультимедиа некоторые производители выпускают специальные **аудио-видеодиски**, которые не совершают термических рекалибровок.

Немного сообразительности, и старая школьная математическая формула для вычисления длины окружности $c=2\pi r$ откроет, что линейная длина внешних дорожек больше, чем длина внутренних. Поскольку все магнитные диски вращаются с постоянной угловой скоростью независимо от того, где находятся головки, возникает очевидная проблема. Раньше при производстве дисков изготовители создавали максимально возможную плотность записи на внутренней дорожке, и при продвижении от центра диска плотность записи постепенно снижалась. Если дорожка содержит, например, 18 секторов, то каждый из них занимает дугу в 20°, и не важно, на каком цилиндре находится эта дорожка

В настоящее время используется другая стратегия. Цилиндры делятся на зоны (на диске их обычно от 10 до 30). При продвижении от центра диска число секторов на дорожке в каждой зоне возрастает. Это изменение усложняет процедуру хранения информации на дорожке, но зато повышает емкость диска, что считается более важным. Все секторы имеют одинаковый размер. К счастью, хоть какие-то вещи в жизни никогда не изменяются.

С диском связан так называемый **контроллер** — микросхема, которая управляет диском. Некоторые контроллеры содержат целый процессор. В задачи контроллера входит получение от программного обеспечения таких команд, как `READ`, `WRITE` и `FORMAT` (то есть запись всех преамбул), управление перемещением рычага, обнаружение и исправление ошибок, преобразование 8-битных байтов, считываемых из памяти, в непрерывный поток битов и наоборот. Некоторые контроллеры производят буферизацию совокупности секторов и кэширование секторов для дальнейшего потенциального использования, а также устраняют поврежденные секторы. Необходимость последней функции вызвана наличием секторов с поврежденным, то есть постоянно намагниченным, участком. Когда контроллер обнаруживает поврежденный сектор, он заменяет его одним из свободных секторов, которые выделяются специально для этой цели в каждом цилиндре или зоне.

Дискеты

С изобретением персонального компьютера появилась необходимость каким-то образом распространять программное обеспечение. Решением этой проблемы послужила **дискета** (**floppy disk** — «гибкий диск»; назван так, потому что первые дискеты были гибкими физически) — небольшой сменный носитель информации. Дискеты были придуманы фирмой IBM. Изначально на них записывалась информация по обслуживанию больших машин (для сотрудников фирмы). Но производители компьютеров вскоре переняли эту идею и стали использовать дискеты в качестве удобного средства записи программного обеспечения и его продажи.

Дискеты обладают теми же общими характеристиками, что и диски, которые мы только что рассматривали, с тем лишь различием, что головки жестких дисков перемещаются над поверхностью диска на воздушной подушке, а у дискет головки касаются поверхности. В результате и сами дискеты, и головки очень быстро изнашиваются. Поэтому когда не происходит считывание и запись информации, головки убираются с поверхности, а компьютер останавливает вращение диска. Это позволяет продлить срок службы дискет. Но при этом, если поступает команда считывания или записи, происходит небольшая задержка (примерно полсекунды) перед тем, как мотор начнет работать.

Существует два вида дискет: 5,25 дюймов и 3,5 дюйма[1]. Каждая из них может быть или с низкой плотностью записи (Low-Density, сокращение LD), или с высокой плотностью записи (High-Density, сокращение HD). Дискеты на 3,5 дюйма выпускаются в жесткой защитной упаковке, поэтому в действительности они не

[1] Дискеты размером 5,25 дюйма уже несколько лет как вышли из обращения. В 2001 году производители персональных компьютеров выпустили стандарт, согласно которому и дискеты размером 3,5 дюйма должны будут окончить свое существование, так как в новые компьютеры не будут устанавливаться приводы для работы с этими дискетами. — *Примеч. научн. ред.*

гибкие. Поскольку 3-дюймовые дискеты вмещают больше данных и лучше защищены от внешних воздействий, они, по существу, заменили старые 5-дюймовые. Наиболее важные параметры всех 4 типов дискет показаны в табл. 2.3.

Таблица 2.3. Параметры четырех видов дискет

Параметры	LD 5,25	HD 5,25	LD 3,5	HD 3,5
Размер, дюймы	5,25	5,25	3,5	3,5
Емкость	360 Кбайт	1,2 Мбайт	720 Кбайт	1,44 Мбайт
Количество дорожек	40	80	80	80
Количество секторов в дорожке	9	15	9	18
Количество головок	2	2	2	2
Число оборотов в мин.	300	360	300	300
Скорость передачи данных, Кбит/с	250	500	250	500
Тип	Гибкий	Гибкий	[1]	Жесткий

Диски IDE

Диски современных персональных компьютеров развились из диска машины IBM PC XT. Это был диск Seagate на 10 Мбайт, управляемый контроллером Xebec на встроенной карте. У этого диска было 4 головки, 306 цилиндров и по 17 секторов на дорожке. Контроллер мог управлять двумя дисками. Операционная система считывала с диска и записывала на диск информацию. Для этого она передавала параметры в регистры процессора и вызывала систему **BIOS (Basic Input Output System — базовую систему ввода-вывода)**, расположенную во встроенном ПЗУ. Система BIOS запрашивала машинные команды для загрузки регистров контроллера, которые начинали передачу данных.

Сначала контроллер помещался на отдельной плате, а позже, начиная с **IDE**-дисков **(Integrated Drive Electronics — устройство со встроенным контроллером)**, которые появились в середине 80-х годов, стал встраиваться в материнскую плату[2]. Однако соглашения о вызовах системы BIOS не изменялись, поскольку необходимо было обеспечить совместимость с более старыми версиями. Обращение к секторам производилось по номерам головки, цилиндра и сектора, причем головки и цилиндры нумеровались с 0, а секторы — с 1. Вероятно, такая ситуация сложилась из-за ошибки одного из программистов BIOS, который написал свой шедевр на ассемблере 8088. Имея 4 бита для номера головки, 6 битов для сектора и 10 битов для цилиндра, диск мог содержать максимум 16 головок, 63 сектора и 1024 цилиндра, то есть всего 1 032 192 сектора. Емкость такого диска составляла 528 Мбайт, и в те времена эта цифра считалась огромной (а вы бы стали сегодня осыпать упреками новую машину, которая не способна работать с дисками объемом более 1 Тбайт?).

[1] Сам магнитный диск — гибкий, жестким является только футляр, в котором он расположен. — *Примеч. научн. ред.*

[2] Встраиваться он стал в сам винчестер, то есть на печатную плату, расположенную в корпусе винчестера. На материнской плате размещается вторая часть контроллера этого интерфейса. — *Примеч. научн. ред.*

Вскоре появились диски объемом более 528 Мбайт, но у них была другая геометрия (4 головки, 32 сектора, 2000 цилиндров). Операционная система не могла обращаться к ним из-за того, что соглашения о вызовах системы BIOS не менялись (требование совместимости). В результате контроллеры начали выдавать ложную информацию, делая вид, что геометрия диска соответствовала системе BIOS. Но на самом деле виртуальная геометрия просто накладывалась на реальную. Хотя этот метод действовал, он затруднял работу операционных систем, которые размещали данные определенным образом, чтобы сократить время поиска.

В конце концов на смену IDE дискам пришли **EIDE**-диски (**Extended IDE** – усовершенствованные IDE), поддерживающие дополнительную схему адресации **LBA (Logical Block Adressing)**, которая просто нумерует секторы от 0 до $2^{24}-1$. Контроллер должен переделывать адреса LBA в адреса головки, сектора и цилиндра, но зато объем диска превышает 528 Мбайт. EIDE диски и контроллеры также имеют другие усовершенствования. Например, они способны контролировать 4 диска вместо двух, у них более высокая скорость передачи данных, и они могут управлять приводом для чтения CD-ROM.

Изначально IDE- и EIDE-диски производились только для систем Intel, поскольку данный интерфейс является точной копией шины IBM PC. Тем не менее в настоящее время некоторые другие компьютеры также используют эти диски из-за их низкой стоимости.

SCSI-диски

SCSI-диски не отличаются от IDE-дисков с точки зрения расположения цилиндров, дорожек и секторов, но они имеют другой интерфейс и более высокую скорость передачи данных. SCSI-диски восходят к изобретателю дискеты Говарду Шугарту (Hovard Shugart). В 1979 году его компания выпустила диск SASI (Shugart Associates System Interface). В 1986 году Институт американских государственных стандартов после длительных обсуждений внес некоторые преобразования в этот диск и изменил его название на **SCSI (Small Computer System Interface – интерфейс малых вычислительных систем)**. Аббревиатура SCSI произносится как «скази». Версии, работающие с более высокой скоростью, получили названия Fast SCSI (10 МГц), Ultra SCSI (20 МГц) и Ultra2 SCSI (40 МГц). Каждая из этих разновидностей также имела 16-битную версию. Основные параметры всех этих версий приведены в табл. 2.4.

Таблица 2.4. Некоторые возможные параметры SCSI

Название	Количество разрядов	Шина (МГц)	Скорость передачи данных по шине, Мбайт/с
SCSI-1	8	5	5
Fast SCSI	8	10	10
Wide Fast SCSI	16	10	20
Ultra SCSI	8	20	20
Wide Ultra SCSI	16	20	40
Ultra2 SCSI	8	40	40
Wide Ultra2 SCSI	16	40	80

Поскольку у дисков SCSI высокая скорость передачи данных, они используются в большинстве рабочих станций UNIX, которые производятся Sun, HP, SGI и другими компаниями. Эти диски также встраиваются в компьютеры Macintosh и сетевые серверы Intel.

SCSI — это не просто интерфейс жесткого диска. Это шина, к которой могут подсоединяться контроллер SCSI и до семи дополнительных устройств. Ими могут быть один или несколько жестких дисков SCSI, компакт-диски, устройства для записи компакт-дисков, сканеры, накопители на магнитной ленте и другие периферийные устройства. Каждое устройство имеет свой идентификационный код от 0 до 7 (до 15 для 16-битных версий). У каждого устройства есть два разъема: один — входной, другой — выходной. Кабели соединяют выходной разъем одного устройства с входным разъемом следующего устройства и т. д. Это похоже на соединение лампочек в елочной гирлянде. Последнее устройство в цепи должно завершать цепь, чтобы отражения от концов шины не искажали другие данные в шине. Обычно контроллер помещается на встроенной карте и является первым звеном цепи, хотя это не обязательно.

Самый обычный кабель для 8-битного SCSI имеет 50 проводов, 25 из которых (заземления) спарены с 25 другими, чтобы обеспечить хорошую помехоустойчивость, которая необходима для высокой скорости работы. Из 25 проводов 8 используются для данных, 1 — для контроля четности, 9 — для управления, а оставшиеся сохраняются для будущего применения. 16-битным и 32-битным устройствам требуется еще 1 кабель для дополнительных сигналов. Кабели могут быть несколько метров в длину, чтобы обеспечивать связь с внешними устройствами (сканерами и т. п.).

Контроллеры и периферийные устройства SCSI могут быть или задатчиками, или приемниками. Обычно контроллер, действующий как задатчик, посылает команды дискам и другим периферийным устройствам, которые, в свою очередь, являются приемниками. Команды представляют собой блоки до 16 байтов, которые сообщают приемнику, что нужно делать. Команды и ответы на них оформляются в виде фраз, при этом используются различные сигналы контроля для разграничения фраз и разрешения конфликтных ситуаций, которые возникают, если несколько устройств одновременно пытаются использовать шину. Это очень важно, так как SCSI позволяет всем устройствам работать одновременно, что сильно повышает производительность среды, поскольку активизируется сразу несколько процессов (в качестве примеров можно привести UNIX или Windows NT). В системах IDE и EIDE если работает одно из устройств, другие не могут действовать одновременно с ним.

RAID-массивы

Производительность процессоров за последнее десятилетие сильно возросла, увеличиваясь почти вдвое каждые 1,5 года. Однако с производительностью дисков дело обстоит иначе. В 70-х годах среднее время поиска в мини-компьютерах составляло от 50 до 100 миллисекунд. Сейчас время поиска составляет около 10 миллисекунд. Во многих отраслях технической промышленности (например, в автомобильной или авиационной) увеличение производительности в 5 или 10 раз за два десятилетия считалось бы грандиозным, но в компьютерной промышленности

эти цифры вызывают недоумение. Таким образом, разрыв между производительностью процессоров и дисков становился все больше и больше.

Как мы уже видели, для того чтобы увеличить скорость работы процессора, используется параллельная обработка данных. Уже на протяжении многих лет разным людям приходит в голову мысль, что было бы неплохо сделать так, чтобы устройства ввода-вывода также могли работать параллельно. В 1988 году Паттерсон, Гибсон и Катс в своей статье предложили 6 разных типов организации дисков, которые могли использоваться для увеличения производительности, надежности или того и другого. Эти идеи были сразу заимствованы производителями компьютеров, что привело к появлению нового класса устройств ввода-вывода под названием **RAID**. Паттерсон, Гибсон и Катс определили RAID как **Redundunt Array of Inexpensive Disks** — «избыточный массив недорогих дисков», но позже буква I в аббревиатуре стала заменять слово Independent (независимый) вместо изначального слова Inexpensive (недорогой). Может быть, в этом случае у производителей появилась возможность выпускать дорогостоящие диски? RAID-массиву противопоставлялся **SLED (Single Large Expensive Disk** — «один большой дорогостоящий диск»).

Основная идея RAID состоит в следующем. Рядом с компьютером (обычно большим сервером) устанавливается бокс с дисками, контроллер диска замещается RAID-контроллером, данные копируются на RAID-массив, а затем производятся обычные действия. Иными словами, операционная система воспринимает RAID как SLED, при этом у RAID-массива выше производительность и надежность. Поскольку SCSI-диски обладают высокой производительностью при довольно низкой цене, при этом один контроллер может управлять несколькими дисками (до 7 дисков на 8-битных моделях SCSI и до 15 на 16-битных), то естественно, что большинство устройств RAID состоит из RAID SCSI-контроллера и бокса SCSI-дисков, которые операционная система воспринимает как один большой диск. Таким образом, чтобы использовать RAID-массив, не требуется никаких изменений в программном обеспечении, что очень выгодно для многих системных администраторов.

Системы RAID имеют несколько преимуществ. Во-первых, как мы уже сказали, программное обеспечение воспринимает RAID как один большой диск. Во-вторых, данные на всех RAID распределены по дискам таким образом, чтобы можно было осуществлять параллельные операции. Несколько различных способов распределения данных были предложены Паттерсоном, Гибсоном и Катсом. Сейчас они известны как RAID-массив нулевого уровня, RAID-массив первого уровня и т. д. до RAID-массива пятого уровня. Кроме того, существует еще несколько уровней, которые мы не будем обсуждать. Термин «уровень» несколько неудачный, поскольку здесь нет никакой иерархической структуры. Просто существует 6 разных типов организации дисков.

RAID-массив нулевого уровня показан на рис. 2.18, *а*. Он представляет собой виртуальный диск, разделенный на полосы, зоны (strips) по k секторов каждая, при этом секторы с 0 по k–1 — полоса 0, секторы с k по 2k–1 — полоса 1 и т. д. Для k=1 каждая полоса — это сектор, для k=2 каждая полоса — это два сектора и т. д. RAID-массив нулевого уровня последовательно записывает полосы по кругу, как показано на рис. 2.18, *а*. На этом рисунке изображен RAID-массив с 4 дисками. Такое распределение данных по нескольким дискам называется **разметкой (striping)**.

Например, если программное обеспечение вызывает команду для считывания блока данных, состоящего из четырех последовательных полосок и начинающегося на границе между полосками, то RAID-контроллер разбивает эту команду на 4 отдельные команды, каждую для одного из четырех дисков, и выполняет их параллельно. Таким образом, мы получаем устройство параллельного ввода-вывода без изменения программного обеспечения.

RAID-массив нулевого уровня лучше всего работает с большими запросами, чем больше запрос, тем лучше. Если полосок в запросе больше, чем дисков в RAID-массиве, то некоторые диски получают по несколько запросов, и как только такой диск завершает выполнение первого запроса, он приступает к следующему. Задача контроллера состоит в том, чтобы разделить запрос должным образом, послать нужные команды соответствующим дискам в правильной последовательности, а затем правильно записать результаты в память. Производительность при таком подходе очень высокая, и осуществить его несложно.

RAID-массив нулевого уровня хуже всего работает с операционными системами, которые время от времени запрашивают данные по одному сектору за раз. В этом случае результаты будут, конечно, правильными, но не будет никакого параллелизма и, следовательно, никакого выигрыша в производительности. Другой недостаток такой структуры состоит в том, что надежность у нее потенциально ниже, чем у SLED. Рассмотрим RAID-массив, состоящий из четырех дисков, на каждом из которых могут происходить сбои в среднем каждые 20 000 часов. Сбои в таком RAID-массиве будут случаться примерно через каждые 5000 часов, при этом все данные могут быть утеряны. У SLED сбои происходят также в среднем каждые 20 000 часов, но так как это один диск, его надежность в 4 раза выше. Поскольку в описанной разработке нет никакой избыточности, это не настоящий[1] RAID-массив.

Следующая разновидность — RAID-массив первого уровня. Он показан на рис. 2.18, *б* и, в отличие от RAID-массива нулевого уровня, является настоящим RAID-массивом[2]. Он дублирует все диски, таким образом получается 4 изначальных диска и 4 резервные копии. При записи информации каждая полоса записывается дважды. При считывании может использоваться любая из двух копий, при этом одновременно может происходить загрузка информации с большего количества дисков, чем у RAID-массива нулевого уровня. Следовательно, производительность при записи будет такая же, как у обычного диска, а при считывании — гораздо выше (максимум в два раза). Отказоустойчивость отличная: если происходит сбой на диске, вместо него используется копия. Восстановление состоит просто в установке нового диска и копировании всей информации с резервной копии на него.

В отличие от нулевого и первого уровней, которые работают с полосами секторов, RAID-массив второго уровня имеет дело со словами, а иногда даже с байтами. Представим, что каждый байт виртуального диска разбивается на два кусочка по 4 бита, затем к каждому из них добавляется код Хэмминга, и таким образом получается слово из 7 битов, у которого 1, 2 и 4 — биты четности. Затем представим, что 7 дисков, изображенные на рис. 2.18, *в*, были синхронизированы по позиции рыча-

[1] На самом деле настоящий, но нулевого уровня. — *Примеч. научн. ред.*

[2] На рис. 2.18, *б* изображен RAID уровня 0+1, а не 1-го уровня. — *Примеч. научн. ред.*

га и позиции вращения. Тогда было бы возможно записать слово из 7 битов с кодом Хэмминга на 7 дисков, по 1 биту на диск.

Подобная схема использовалась в так называемых думающих машинах СМ-2. К 32-битному слову с данными добавлялось 6 битов четности (код Хэмминга). В результате получалось 38-битное кодированное слово, к которому добавлялся дополнительный бит четности, и это слово записывалось на 39 дисков. Общая производительность была огромной, так как одновременно могло записываться 32 сектора данных. При утрате одного из дисков проблем также не возникало, поскольку потеря одного диска означала потерю одного бита в каждом 39-битном слове, а с этим код Хэмминга справлялся моментально.

С другой стороны, эта схема требует, чтобы все диски были синхронизированы по вращению. Кроме того, она имеет смысл, только если имеется достаточно большое количество дисков (даже при наличии 32 дисков для данных и 6 дисков для битов четности накладные расходы составляют 19 процентов). К тому же требуется большая работа контроллера, поскольку он должен вычислять контрольную сумму кода Хэмминга каждый раз при передаче бита.

RAID-массив третьего уровня представляет собой упрощенную версию RAID-массива второго уровня. Он изображен на рис. 2.18, *г*. Здесь для каждого слова данных вычисляется 1 бит четности и записывается на диск четности. Как и в RAID-массиве второго уровня, диски должны быть точно синхронизированы, поскольку каждое слово данных распределено на несколько дисков.

На первый взгляд может показаться, что один бит четности только обнаруживает, но не исправляет ошибки. Если речь идет о случайных необнаруженных ошибках, это наблюдение верно. Однако если речь идет о сбое на диске, бит четности обеспечивает исправление 1-битной ошибки, поскольку позиция неправильного бита известна. Если происходит сбой, контроллер выдает информацию, что все биты равны 0. Если в слове возникает ошибка четности, бит с диска, на котором произошел сбой, должен быть 1, и следовательно, он исправляется. Хотя RAID-массивы второго и третьего уровней обеспечивают очень высокую скорость передачи данных, число запросов устройств ввода-вывода в секунду не больше, чем при наличии одного диска.

RAID-массиавы четвертого и пятого уровней опять работают с полосами, а не со словами с битами четности, и не требуют синхронизации дисков. RAID-массив четвертого уровня (см. рис. 2.18, *д*) устроен так же, как RAID-массив нулевого уровня, с тем различием, что у RAID-массива четвертого уровня имеется дополнительный диск, на который записываются полосы четности. Например, пусть каждая полоса состоит из k байтов. Все полосы должны находиться в отношении «исключающего ИЛИ», и полоса четности для проверки этого отношения также должна состоять из k байтов. Если происходит сбой на диске, утраченные байты могут быть вычислены заново при использовании информации с диска четности.

Такая разработка предохраняет от потерь на диске, но обладает очень низкой производительностью в случае небольших исправлений. Если изменяется 1 сектор, необходимо считывать информацию со всех дисков, для того чтобы опять вычислить четность, которая должна быть записана заново. Вместо этого можно считать с диска прежние данные и прежнюю четность и из них вычислить новую четность. Но даже с такой оптимизацией процесса при наличии небольших исправлений требуется произвести два считывания и две записи.

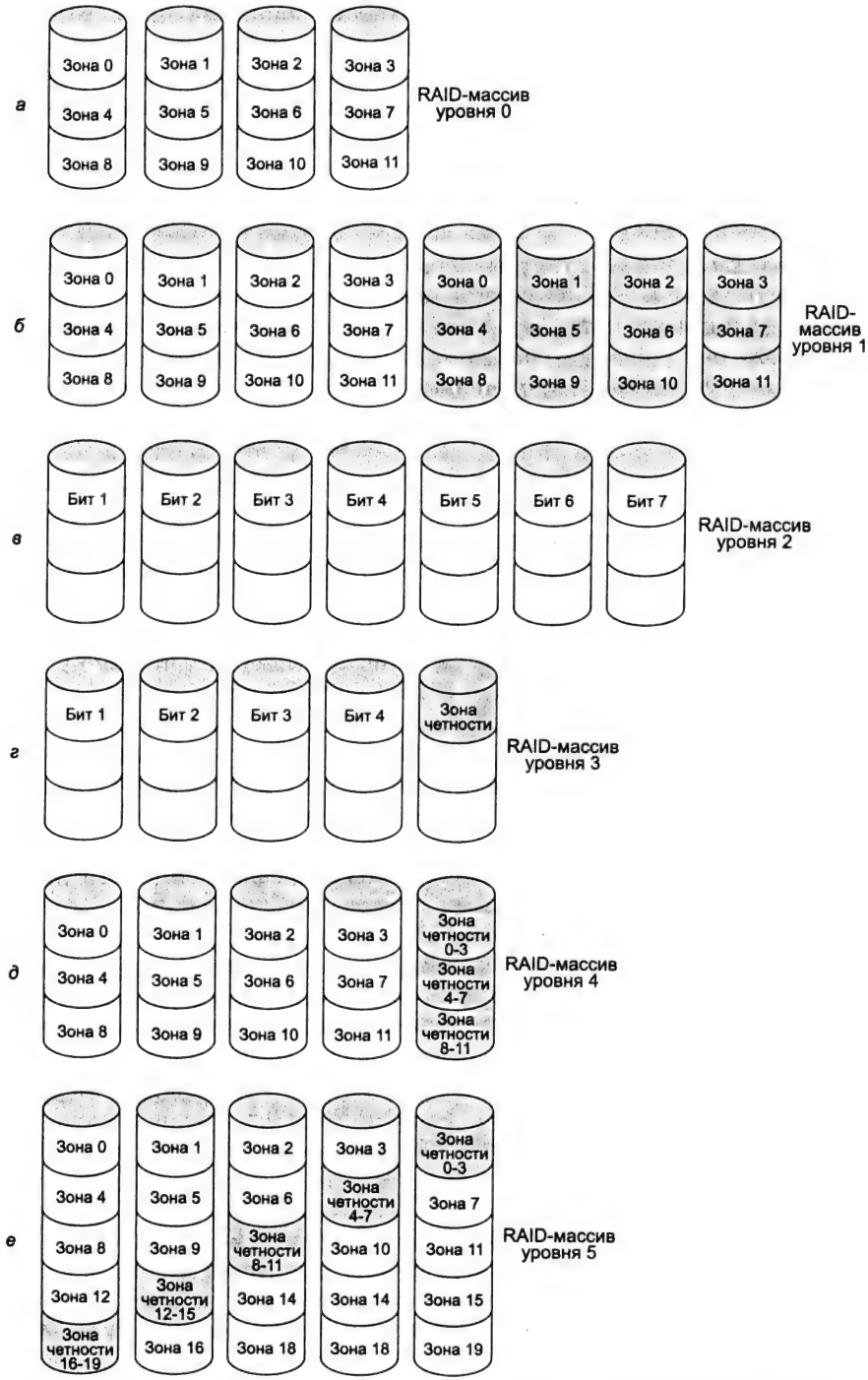


Рис. 2.18. RAID-массивы с нулевого по пятый уровень. Резервные копии и диски четности закрашены серым цветом

Такие трудности при загрузке на диск четности могут быть препятствием высокой производительности. Эта проблема устраняется в RAID-массиве пятого уровня, в котором биты четности распределяются равномерно по всем дискам и записываются по кругу, как показано на рис. 2.18, *е*. Однако в случае сбоя на диске восстановить содержание утраченного диска достаточно сложно, хотя и возможно.

Компакт-диски

В последние годы помимо магнитных дисков стали доступны оптические диски. Они обладают более высокой плотностью записи, чем обычные магнитные диски[1]. Оптические диски изначально использовались для записи телевизионных программ, но позже они стали использоваться как средства хранения информации в компьютерной технике. Из-за потенциально огромной емкости оптические диски стали предметом многих исследований, и их усовершенствование происходило довольно быстро.

Первые оптические диски были изобретены голландской корпорацией Philips для хранения кинофильмов. Они имели 30 см в диаметре, выпускались под маркой LaserVision, но нигде, кроме Японии, не пользовались популярностью.

В 1980 году корпорация Philips вместе с Sony разработала CD (Compact Disc — компакт-диск), который быстро вытеснил виниловые диски, использовавшиеся для музыкальных записей.

Описание технических деталей компакт-диска было опубликовано в официальном Международном Стандарте (IS 10149), который часто называют **Красной книгой** (по цвету обложки). Международные Стандарты издаются Международной Организацией Стандартизации, которая представляет собой аналог таких национальных групп стандартизации, как ANSI, DIN и т. п. У каждой такой группы есть свой IS-номер (International Standard — Международный Стандарт). Международный Стандарт технических характеристик диска был опубликован для того, чтобы компакт-диски от разных музыкальных издателей и проигрыватели от разных производителей стали совместимыми. Все компакт-диски должны быть 120 мм в диаметре и 1,2 мм в толщину, а диаметр отверстия в середине должен составлять 15 мм. Аудио-компакт-диски были первым средством хранения цифровой информации, которое вышло на массовый рынок потребления. Предполагается, что они будут использоваться на протяжении ста лет. Пожалуйста, сравните в 2080 году работу самой последней разработки и первой партии компакт-дисков.

Компакт-диск изготавливается с использованием очень мощного инфракрасного лазера, который выжигает отверстия диаметром 0,8 микрон в специальном стеклянном контрольном диске. По этому контрольному диску делается шаблон с выступами в тех местах, где лазер прожег отверстия. В шаблон вводится жидкая смола (поликарбонат), и таким образом получается компакт-диск с тем же набором отверстий, что и в стеклянном диске. На смолу наносится очень тонкий слой алюминия, который в свою очередь покрывается защитным лаком. После этого наклеивается этикетка. Углубления в нижнем слое смолы в английском языке называются термином **«впадина»** (pit), а ровные пространства между впадинами называются термином **«площадка»** (land).

[1] Это утверждение ошибочно. — *Примеч. научн. ред.*

Во время воспроизведения лазерный диод небольшой мощности светит инфракрасным светом с длиной волны 0,78 микрон на сменяющиеся впадины и площадки. Лазер находится на той стороне диска, где слой смолы, поэтому впадины для лазера оказываются выступами на ровной поверхности. Так как впадины имеют высоту в четверть длины волны света лазера, длина волны света, отраженного от впадины, составляет половину длины волны света, отраженного от окружающей выступ ровной поверхности. В результате, если свет отражается от выступа, фотодетектор проигрывателя получает меньше света, чем при отражении от площадки. Именно таким образом проигрыватель отличает впадину от площадки. Хотя, казалось бы, проще всего использовать впадину для записи 0, а площадку для записи 1, более надежно использовать переход впадина/площадка или площадка/впадина для 1 и его отсутствие для 0.

Впадины и площадки записываются по спирали. Запись начинается на некотором расстоянии от отверстия в центре диска и продвигается к краю, занимая 32 мм диска. Спираль проходит 22 188 оборотов вокруг диска (примерно 600 на 1 мм). Если ее распрямить, ее длина составит 5,6 км. Спираль изображена на рис. 2.19.

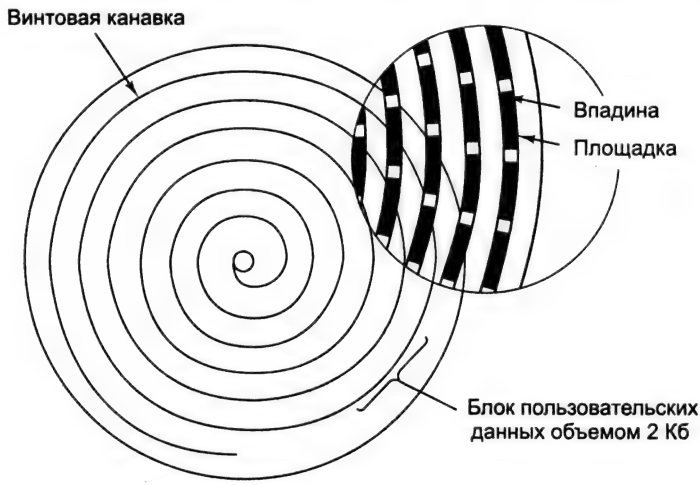


Рис. 2.19. Структура записи компакт-диска

Чтобы музыка звучала нормально, впадины и площадки должны сменяться с постоянной линейной скоростью. Следовательно, скорость вращения компакт-диска должна постепенно снижаться по мере продвижения считывающей головки от центра диска к внешнему краю. Когда головка находится на внутренней стороне диска, скорость вращения составляет 530 оборотов в минуту, чтобы достичь желаемой скорости 120 см/с. Когда головка находится на внешней стороне диска, скорость вращения падает до 200 оборотов в минуту, чтобы обеспечить такую же линейную скорость. Диск с постоянной линейной скоростью отличается от магнитного диска, который работает с постоянной угловой скоростью, независимо от того, где находится головка в настоящий момент. Кроме того, скорость вращения компакт-диска (530 оборотов в минуту) очень сильно отличается от скорости вращения магнитных дисков, которая составляет от 3600 до 7200 оборотов в минуту.

В 1984 году Philips и Sony начали использовать компакт-диски для хранения компьютерных данных. Они опубликовали **Желтую книгу**, в которой определили точный стандарт того, что они назвали **CD-ROM (Compact Disc — Read Only Memory — компакт-диск — постоянное запоминающее устройство)**. Чтобы влиться в широко развернувшийся к тому времени рынок аудио-компакт-дисков, компьютерные компакт-диски должны были быть такого же размера, как аудио-диски, механически и оптически совместимыми с ними и производиться по той же технологии. Вследствие такого решения потребовались моторы, работающие с низкой скоростью и способные менять скорость. Стоимость производства одного компакт-диска составляла в среднем меньше 1 доллара.

В Желтой книге определено форматирование компьютерных данных. В ней также описаны усовершенствованные приемы исправления ошибок, что является существенным шагом, поскольку компьютерщики, в отличие от любителей музыки, придают очень большое значение ошибкам в битах. Разметка компакт-диска состоит в кодировании каждого байта 14-битным символом. Как мы видели выше, 14 битов достаточно для того, чтобы закодировать кодом Хэмминга 8-битный байт, при этом останется два лишних бита. На самом деле используется более мощная система кодировки. Перевод из 16-битной в 8-битную систему для считывания информации производится аппаратным обеспечением с помощью поисковых таблиц.

На следующем уровне 42 последовательных символа формируют **фрейм** из 588 битов. Каждый фрейм содержит 192 бита данных (24 байта). Оставшиеся 396 битов используются для исправления ошибок и контроля. У аудио- и компьютерных компакт-дисков эта система одинакова.

У компьютерных компакт-дисков каждые 98 фреймов группируются в **сектор**, как показано на рис. 2.20. Каждый сектор начинается с преамбулы из 16 байтов, первые 12 из которых — 00FFFFFFFFFFFFFFFFFF00 (в шестнадцатеричной системе), что дает возможность проигрывателю определять начало сектора. Следующие 3 байта содержат номер сектора, который необходим, поскольку поиск на компакт-диске, на котором данные записаны по спирали, гораздо сложнее, чем на магнитном диске, где данные записаны на концентрических дорожках. Чтобы найти определенный сектор, программное обеспечение подсчитывает, куда приблизительно нужно направляться; туда помещается считывающая головка, а затем начинается поиск преамбулы, чтобы установить, насколько верен был подсчет. Последний байт преамбулы определяет тип диска.



Рис. 2.20. Схема расположения данных на компакт-диске

Желтая книга определяет 2 типа дисков. Тип 1 использует расположение данных, показанное на рис. 2.20, где преамбула составляет 16 байтов, данные — 2048 байтов, а код с исправлением ошибок — 228 байтов (код Рида—Соломона). Тип 2 объединяет данные и коды с исправлением ошибок в поле данных на 2336 байтов. Такая схема применяется для приложений, которые не нуждаются в исправлении ошибок (или, точнее, которые не могут выделить время для этого), например аудио и видео. Отметим, что для обеспечения высокой степени надежности используются три схемы исправления ошибок: в пределах символа, в пределах фрейма и в пределах сектора. Одиночные ошибки в битах исправляются на самом нижнем уровне, пакеты ошибок — на уровне фреймов, а все остаточные ошибки — на уровне секторов. Для обеспечения такой надежности необходимо 98 фреймов по 588 битов (7203 байта), чтобы поддерживать 2048 байтов полезной нагрузки. Таким образом, эффективность составляет всего 28%.

Односкоростные устройства для чтения компакт-дисков считывают 75 секторов/с, что обеспечивает скорость передачи данных 153 600 байт/с при диске первого типа и 175 200 байт/с при диске второго типа. Двухскоростные устройства работают в два раза быстрее и т. д., до самой высокой скорости. Стандартный аудио-компакт-диск располагает емкостью для 74 минут музыки, что соответствует 681 984 000 байтов. Это число равно 650 Мбайт, так как 1 Мбайт=2^{20} байтов (1 048 576 байт), а не 1 000 000 байтов.

Отметим, что даже устройство для чтения компакт-дисков со скоростью, обозначаемой как 32x (4 915 200 байт/с), не сравнимо с быстрым магнитным диском SCSI-2 (10 Мбайт/с), несмотря на то, что многие устройства для чтения компакт-дисков используют интерфейс SCSI (кроме того, применяется интерфейс EIDE). Когда вы понимаете, что время поиска составляет несколько сотен миллисекунд, становится ясно, что устройства для чтения компакт-дисков по производительности сильно уступают магнитным дискам, хотя емкость компакт-дисков гораздо выше[1].

В 1986 году корпорация Philips опубликовала **Зеленую книгу**, добавив графику и возможность помещать аудио-, видео- и обычные данные в одном секторе, что было необходимо для мультимедийных компакт-дисков.

Последняя проблема, которую нужно было разрешить при разработке компакт-дисков, — совместимость файловой системы. Чтобы можно было использовать один и тот же компакт-диск на разных компьютерах, необходимо было соглашение по поводу файловой системы компакт-дисков. Чтобы выпустить такое соглашение, представители разных компьютерных компаний встретились на озере Тахо в Хай-Сьерраз (the High Sierras) на границе Калифорнии и Невады и разработали файловую систему, которую они назвали **High Sierra**. Позднее эта система превратилась в Международный Стандарт (IS 9660). Существует три уровня этого стандарта. На первом уровне допустимы имена файлов до 8 символов, за именем файла может следовать расширение до трех символов (соглашение для наименования файлов в MS-DOS). Имена файлов могут содержать только прописные буквы, цифры и символ подчеркивания. Директории могут вкладываться одна в другую, причем

[1] Емкость компакт-дисков на два порядка ниже емкости современных магнитных дисков. — *Примеч. научн. ред.*

допускается не более 8 иерархических ступеней. Имена директорий могут не содержать расширения. На первом уровне требуется, чтобы все файлы были смежными, что не представляет особых трудностей в случае с носителем, на который информация записывается только один раз. Любой компакт-диск, который соответствует Международному Стандарту IS 9660 первого уровня, может быть прочитан с использованием системы MS-DOS, компьютеров Apple, Unix и практически любого другого компьютера. Производители компакт-дисков считают это свойство большим плюсом.

Второй уровень Международного Стандарта IS 9660 допускает имена файлов до 32 символов, а на третьем уровне допускается несмежное расположение файлов. Расширения Rock Ridge (названные так причудливо в честь города в фильме Джина Уайлдера «Горящие седла») допускают очень длинные имена файлов (для Unix), UID, GID и символические связи, но компакт-диски, не соответствующие первому уровню, не будут читаться на всех компьютерах.

Компакт-диски стали очень популярны для распространения компьютерных игр, художественных фильмов, энциклопедий, атласов и различного рода справочников. В настоящее время на компакт-дисках выпускается бóльшая часть коммерческого программного обеспечения. Сочетание большой вместимости и низкой цены делает компакт-диски подходящими для бесчисленного множества приложений.

CD-R

Вначале оборудование, необходимое для изготовления контрольных компакт-дисков (как аудио-, так и компьютерных), было очень дорогим. Но, как это обычно происходит в компьютерной промышленности, ничего не остается дорогим долгое время. К середине 90-х годов записывающие устройства для компакт-дисков размером не больше проигрывателя стали обычными и общедоступными, их можно было приобрести в любом магазине компьютерной техники. Эти устройства все еще отличались от магнитных дисков, поскольку информацию, записанную однажды на компакт-диск, уже нельзя было стереть. Тем не менее они быстро нашли сферу применения в качестве дополнительных носителей информации, а основными носителями продолжали служить жесткие диски. Кроме того, отдельные лица и начинающие компании могли выпускать свои собственные компакт-диски небольшими партиями или производить контрольные диски и отправлять их на крупные коммерческие предприятия, занимающиеся изготовлением копий. Такие диски называются **CD-R (CD-Recordable)**.

CD-R производится на основе поликарбонатных заготовок. Такие же заготовки используются при производстве компакт-дисков. Однако диски CD-R отличаются от компакт-дисков тем, что CD-R содержат канавку шириной 0,6 мм, чтобы направлять лазер при записи. Канавка имеет синусоидальное отклонение 0,3 мм на частоте ровно 22,05 кГц для обеспечения постоянной обратной связи, чтобы можно было точно определить скорость вращения и в случае необходимости отрегулировать ее. CD-R выглядит как обычный диск, только он не серебристого, а золотистого цвета, так как для изготовления отражающего слоя вместо алюминия

используется настоящее золото. В отличие от обычных компакт-дисков с физическими углублениями, CD-R моделируются с помощью изменения отражательной способности впадин и площадок. Для этого между слоем поликарбоната и отражающим слоем золота помещается слой красителя, как показано на рис. 2.21. Используется два вида красителей: цианин зеленого цвета и пталоцианин желтовато-оранжевого цвета. Химики могут спорить до бесконечности, какой из них лучше. Эти красители сходны с теми красителями, которые используются в фотографии, и именно поэтому Kodak и Fuji являются главными производителями дисков CD-R.

На начальной стадии слой красителя прозрачен, что дает возможность свету лазера проходить сквозь него и отражаться от слоя золота. При записи информации мощность лазера увеличивается до 8–16 мВт. Когда луч достигает красителя, краситель нагревается, и в результате разрушается химическая связь. Такое изменение молекулярной структуры создает темное пятно. При чтении (когда мощность лазера составляет 0,5 мВт) фотодетектор улавливает разницу между темными пятнами, где краситель был поврежден, и прозрачными областями, где краситель не тронут. Это различие воспринимается как различие между впадинами и площадками даже при чтении на обычном устройстве для считывания компакт-дисков или на аудио-проигрывателе.

Ни один новый вид компакт-дисков не обходился без публикации параметров в книге определенного цвета. В случае с CD-R это была **Оранжевая книга**, вышедшая в 1989 году. Этот документ определяет диск CD-R, а также новый формат, **CD-ROM XA**, который позволяет записывать информацию на CD-R постепенно: несколько секторов сегодня, несколько секторов завтра, несколько секторов через месяц. Группа последовательных секторов, записываемых за 1 раз, называется **дорожкой компакт-диска**.

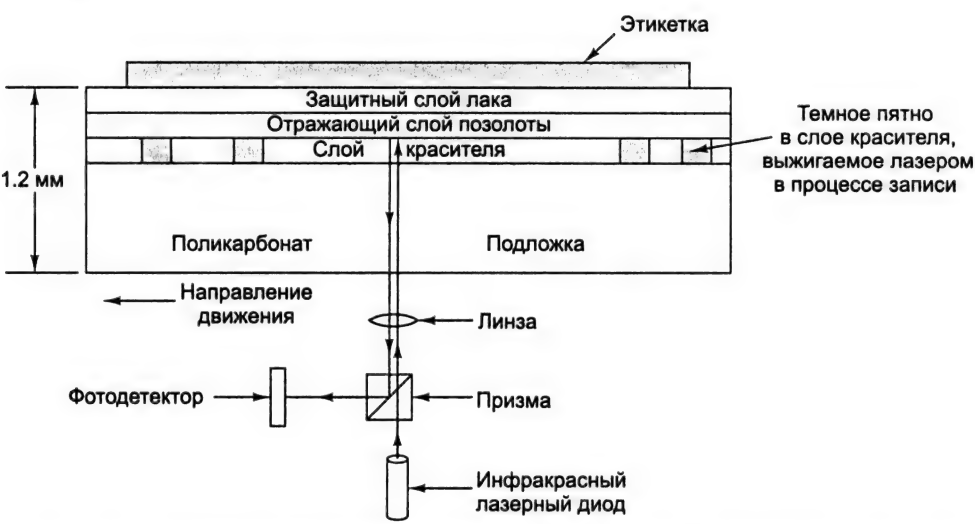


Рис. 2.21. Поперечное сечение диска CD-R и лазера (масштаб не соблюдается). Обычный компакт-диск имеет сходную структуру, но у него отсутствует слой красителя и вместо слоя золота используется слой алюминия с выемками

Одним из первых применений CD-R был фото-компакт-диск фирмы Kodak. При такой системе клиент приносит экспонированную пленку и старый фото-компакт-диск в проявочную машину и получает свой старый компакт-диск, на который после старых снимков записаны новые. Новый пакет данных, полученный в результате сканирования негативов, записывается на компакт-диск в виде отдельной дорожки. Такой способ записи необходим, поскольку заготовки дисков CD-R слишком дорого стоят, поэтому записывать каждую новую пленку на новый диск невыгодно.

Однако с появлением такого типа записи возникла новая проблема. До появления Оранжевой книги у всех компакт-дисков был единый **VTOC (Volume Table of Contents — оглавление диска)**. При такой системе дозаписывать диск было невозможно. Решением данной проблемы стало предложение давать для каждой дорожки компакт-диска отдельный VTOC. В список файлов VTOC могут включаться все файлы из предыдущих дорожек или некоторые из них. После того как диск CD-R вставлен в считывающее устройство, операционная система начинает искать среди дорожек самый последний VTOC, который выдает текущее состояние диска. Если в текущий VTOC включить только некоторые, а не все файлы из предыдущих дорожек, может создаться впечатление, что файлы были удалены. Дорожки можно группировать в **сессии**. В этом случае мы говорим о **многосессионных** компакт-дисках. Стандартные аудио-проигрыватели не могут работать с многосекционными компакт-дисками, поскольку они ожидают единый VTOC в начале диска.

Каждая дорожка должна записываться непрерывно без остановок. Поэтому жесткий диск, от которого поступают данные, должен работать достаточно быстро, чтобы вовремя их доставлять. Если файлы, которые нужно скопировать, расположены в разных частях жесткого диска, длительное время поиска может послужить причиной остановки потока данных на CD-R и, следовательно, причиной недобора данных буфера. В результате недобора данных буфера у вас появится замечательная блестящая (правда, немного дорогая) подставка для стаканов и бутылок. Программное обеспечение CD-R обычно предлагает параметр сбора всех необходимых файлов в виде блока последовательных данных. То есть до передачи файлов на CD-R создается копия компакт-диска в 650 Мбайт. Однако этот процесс обычно удваивает время записи, требует наличия 650 Мбайт свободного дискового пространства и не защищает от того, что жесткие диски начинают совершать рекалибровку в случае перегрева.

С появлением CD-R у отдельных лиц и компаний появилась возможность без труда копировать компьютерные и музыкальные компакт-диски, что часто происходит с нарушением авторских прав. Были придуманы разные средства, препятствующие производству пиратской продукции и затрудняющие чтение компакт-дисков с помощью программного обеспечения, разработанного не производителем данного диска. Один из таких способов — запись на компакт-диск информации о том, что длина всех файлов составляет несколько гигабайт. Это препятствует копированию файлов на жесткий диск с использованием обычного программного обеспечения. Настоящие размеры файлов включаются в программное обеспечение производителя данного компакт-диска или прячутся где-нибудь на компакт-диске (часто в зашифрованном виде). При другом подходе в избранные секторы вставляются заведомо неправильные коды с исправлением ошибок. Программ-

ное обеспечение, прилагаемое к данному компакт-диску, зафиксирует эти ошибки, а обычное программное обеспечение проверит эти коды с исправлением ошибок и не будет работать, если они заведомо правильные. Кроме того, возможно использование нестандартных промежутков между дорожками и других физических «дефектов».

CD-RW

Хотя люди и привыкли к таким носителям информации, которые нельзя перезаписывать (такими носителями являются, например, бумага или фотопленка), все равно существует спрос на перезаписываемые компакт-диски. В настоящее время появилась технология **CD-RW (CD-ReWritable — перезаписываемый компакт-диск)**. При этом используется носитель такого же размера, как и CD-R. Однако вместо красителя (цианина или пталоцианина) при производстве CD-RW используется сплав серебра, индия, сурьмы и теллура для записывающего слоя. Этот сплав имеет два состояния: кристаллическое и аморфное, которые обладают разной отражательной способностью.

Устройства для записи компакт-дисков снабжены лазером с тремя вариантами мощности. При самой высокой мощности лазер расплавляет сплав, переводя его из кристаллического состояния с высокой отражательной способностью в аморфное состояние с низкой отражательной способностью, так получается впадина. При средней мощности сплав расплавляется и возвращается обратно в естественное кристаллическое состояние, при этом впадина превращается снова в площадку. При низкой мощности лазер определяет состояние материала (для считывания информации), никакого перехода состояний при этом не происходит.

CD-RW не заменили CD-R, поскольку заготовки дисков CD-RW гораздо дороже заготовок CD-R. Кроме того, для приложений, поддерживающих жесткие диски, большим плюсом является тот факт, что с CD-R нельзя случайно стереть информацию.

DVD

Основной формат компакт-дисков использовался с 1980 года. С тех пор технологии продвинулись вперед, поэтому оптические диски с высокой емкостью сейчас вполне доступны по цене и пользуются большим спросом. Голливуд с радостью заменил бы аналоговые видеозаписи на цифровые диски, поскольку они лучше по качеству, их дешевле производить, они дольше служат, занимают меньше места на полке в магазине и их не нужно перематывать. Компании, выпускающие бытовую технику, занимаются поисками нового массового продукта, а многие компьютерные компании хотят добавить к своему программному обеспечению мультимедиа.

Такое развитие технологий и спроса на продукцию трех чрезвычайно богатых и мощных индустрий привело к появлению **DVD** (изначально сокращение от **Digital Video Disk — цифровой видеодиск**, а сейчас официально **Digital Versatile Disk — цифровой универсальный диск**). Диски DVD в целом похожи на компакт-диски. Как и обычные компакт-диски, они имеют 120 мм в диаметре, создаются на основе поликарбоната и содержат впадины и площадки, которые освеща-

ются лазерным диодом и считываются фотодетектором. Однако существует несколько различий:

1. Впадины меньшего размера (0,4 микрона вместо 0,8 микрона, как у обычного компакт-диска).
2. Более плотная спираль (0,74 микрона между дорожками вместо 1,6 микрона).
3. Красный лазер (с длиной волны 0,65 микрона вместо 0,78 микрона).

   В совокупности эти усовершенствования дали семикратное увеличение емкости (до 4,7 Гбайт). Считывающее устройство для DVD 1x работает со скоростью 1,4 Мбайт/с (скорость работы считывающего устройства для компакт-дисков составляет 150 Кбайт/с). К несчастью, из-за перехода к красному лазеру потребовались DVD-проигрыватели с двумя лазерами или со сложной оптической системой, чтобы можно было читать существующие музыкальные и компьютерные компакт-диски. Таким образом, не все DVD-проигрыватели могут работать со старыми компакт-дисками. Кроме того, не всегда возможно считывание дисков CD-R и CD-RW.

Достаточно ли 4,7 Гбайт? Может быть. Если использовать сжатие MPEG-2 (стандарт IS 13346), DVD-диск объемом 4,7 Гбайт может вместить полноэкранную видеозапись на 133 минуты с высокой разрешающей способностью (720×480) вместе с озвучиванием на 8 языках и субтитрами на 32 других языках. Около 92% фильмов, снятых в Голливуде, по длительности меньше 133 минут. Тем не менее для некоторых приложений (например, игр мультимедиа или справочных изданий) может понадобиться больше места, а Голливуд мог бы записывать по несколько фильмов на один диск. Поэтому было разработано 4 формата:

1. Односторонние однослойные (4,7 Гбайт).
2. Односторонние двуслойные (8,5 Гбайт).
3. Двусторонние однослойные (9,4 Гбайт).
4. Двусторонние двуслойные (17 Гбайт).

Зачем так много форматов? Если говорить коротко, основная причина — убеждения компаний. Philips и Sony считали, что нужно выпускать односторонние диски с двойным слоем, а Toshiba и Time Warner хотели производить двусторонние диски с одним слоем. Philips и Sony думали, что покупатели не захотят переворачивать диски, а компания Time Warner полагала, что если поместить два слоя на одну сторону диска, он не будет работать. Компромиссное решение — выпускать все варианты, а рынок уже сам определит, какой из вариантов выживет.

При двуслойной технологии на нижний отражающий слой помещается полуотражающий слой. В зависимости от того, где фокусируется лазер, он отражается либо от одного слоя, либо от другого. Чтобы обеспечить надежное считывание информации, впадины и площадки нижнего слоя должны быть немного больше по размеру, поэтому его емкость немного меньше, чем у верхнего слоя.

Двусторонние диски создаются путем склеивания двух односторонних дисков по 0,6 мм. Чтобы толщина всех версий была одинаковой, односторонний диск толщиной 0,6 мм приклеивается к пустой подложке (возможно, в будущем эта под-

ложка будет содержать 133 минуты рекламы, в надежде, что покупатели заинтересуются, что там на нем). Структура двустороннего диска с двойным слоем показана на рис. 2.22.

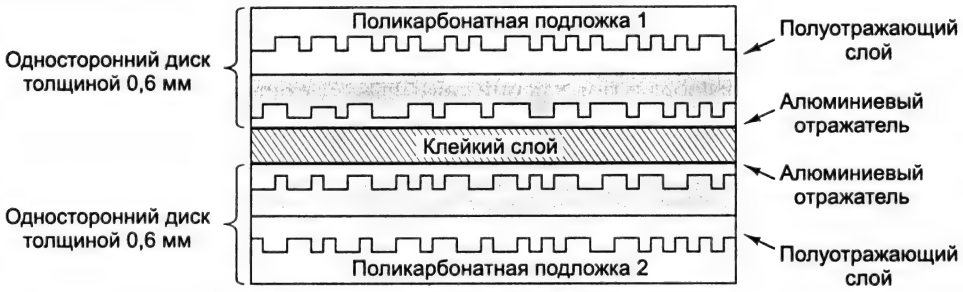


Рис. 2.22. Двусторонний диск DVD с двойным слоем

DVD был разработан корпорацией, состоящей из 10 компаний по производству бытовой техники, семь из которых были японскими, в тесном сотрудничестве с главными студиями Голливуда (японские компании являлись владельцами некоторых из этих студий). Ни компьютерная, ни телекоммуникационная промышленность не были вовлечены в разработку, и в результате упор был сделан на использование DVD для видеопрокатов и распродаж. Перечислим некоторые стандартные особенности DVD: возможность исключать непристойные сцены из фильма (чтобы родители могли превращать фильм типа NC17[1] в фильм, который можно смотреть детям), шестиканальный звук, поддержка для перемасштабирования. Последняя особенность позволяет DVD-проигрывателю решать, как обрезать правый и левый край фильмов (у которых соотношение ширины и высоты 3:2) так, чтобы они подходили к современным телевизорам (с форматом 4:3).

Еще одна особенность, которая, вероятно, никогда не пришла бы в голову разработчикам компьютерных технологий, — намеренная несовместимость дисков для Соединенных Штатов и для европейских стран и другие стандарты для других континентов. Голливуд ввел такую систему, потому что новые фильмы всегда сначала выпускаются на экраны в Соединенных Штатах и только после появления видеокассет отправляются в Европу. Это делается для того, чтобы европейские магазины видеопродукции не могли покупать видеозаписи в Америке слишком рано (вследствие этого мог сократиться объем продаж новых фильмов в Европе). Если бы Голливуд стоял во главе компьютерной промышленности, то в Америке были бы дискеты 3,5 дюйма, а в Европе — 9 см.

Поскольку DVD-диски пользуются большой популярностью, возможно, что в скором времени диски DVD-R (на которых возможна запись информации) и DVD-RW (на которых возможна перезапись информации) станут продуктами массового потребления. Однако успех DVD не гарантирован, поскольку кабельные компании планируют доставлять фильмы несколько другим способом — по кабелю, и борьба уже началась.

[1] NC17 — фильмы, содержащие сцены секса и насилия и не предназначенные для просмотра детьми. — *Примеч. перев.*

Процесс ввода-вывода

Как мы сказали в начале этой главы, компьютерная система состоит из трех основных компонентов: центрального процессора, памяти (основной и вспомогательной) и устройств ввода-вывода (принтеров, сканеров и модемов). До сих пор мы рассматривали центральные процессоры и память. Теперь мы займемся изучением устройств ввода-вывода и тем, как они связываются с остальными компонентами системы.

Шины

Большинство персональных компьютеров и рабочих станций имеют физическую структуру, сходную с той, которая изображена на рис. 2.23. Обычное устройство представляет собой металлический корпус с большой интегральной схемой на дне, которая называется **материнской платой**. Материнская плата содержит микросхему процессора, несколько разъемов для модулей DIMM и различные микросхемы поддержки. Она также содержит шину, протянутую вдоль нее, и несколько разъемов для подсоединения плат устройств ввода-вывода. Иногда может быть две шины: одна с высокой скоростью передачи данных (для современных плат устройств ввода-вывода), а другая с низкой скоростью передачи данных (для старых плат устройств ввода-вывода).

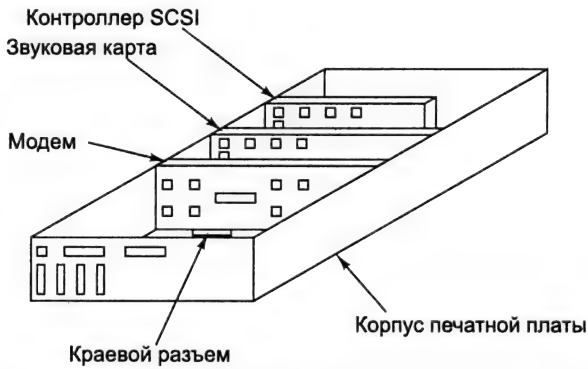


Рис. 2.23. Физическая структура персонального компьютера

Логическая структура обычного персонального компьютера показана на рис. 2.24. У данного компьютера имеется одна шина для соединения центрального процессора, памяти и устройств ввода-вывода, однако большинство систем содержат две и более шин. Каждое устройство ввода-вывода состоит из двух частей: одна из них содержит большую часть электроники и называется **контроллером**, а другая представляет собой само устройство ввода-вывода, например дисковод. Контроллер обычно содержится на плате, которая втыкается в свободный разъем. Исключение представляют контроллеры, являющиеся обязательными (например, клавиатура), которые иногда располагаются на материнской плате. Хотя дисплей (монитор) и не является факультативным устройством, соответствующий контроллер иногда рас-

полагается на встроенной плате, чтобы пользователь мог по желанию выбирать платы с графическими ускорителями или без них, устанавливать дополнительную память и т. д. Контроллер связывается с самим устройством кабелем, который подсоединяется к разъему на задней стороне корпуса.

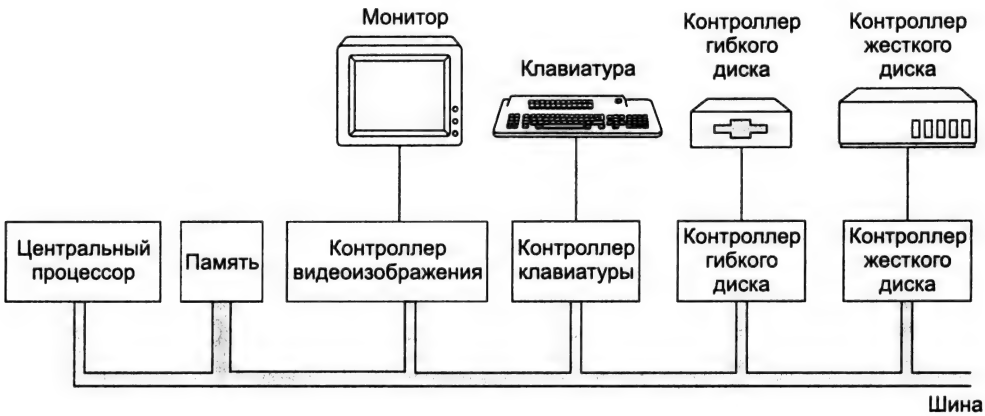


Рис. 2.24. Логическая структура обычного персонального компьютера

Контроллер управляет своим устройством ввода-вывода и регулирует доступ к шине для этого. Например, если программа запрашивает данные с диска, она посылает команду контроллеру диска, который затем отправляет команды поиска и другие команды на диск. После нахождения соответствующей дорожки и сектора диск начинает передавать контроллеру данные в виде потока битов. Задача контроллера состоит в том, чтобы разбить поток битов на куски и записывать каждый такой кусок в память по мере их накопления. Отдельный кусок обычно представляет собой одно или несколько слов. Если контроллер считывает данные из памяти или записывает их в память без участия центрального процессора, то говорят, что осуществляется **прямой доступ к памяти** (**Direct Memory Access**, сокращенно **DMA**). Когда передача данных заканчивается, контроллер вызывает **прерывание**, вынуждая центральный процессор приостановить работу текущей программы и начать выполнение особой процедуры. Эта процедура называется **программой обработки прерывания** и нужна, чтобы проверить ошибки, произвести необходимые действия в случае их обнаружения и сообщить операционной системе, что процесс ввода-вывода завершен. Когда программа обработки прерывания завершена, процессор возобновляет работу программы, которая была приостановлена в момент прерывания.

Шина используется не только контроллерами ввода-вывода, но и процессором для передачи команд и данных. А что происходит, если процессор и контроллер ввода-вывода хотят получить доступ к шине одновременно? В этом случае особая микросхема, которая называется **арбитром шины**, решает, чья очередь первая. Обычно предпочтение отдается устройствам ввода-вывода, поскольку работу дисков и других движущихся устройств нельзя прерывать, так как это может привести к потере данных. Когда ни одно устройство ввода-вывода не функционирует, центральный процессор может полностью распоряжаться шиной для связи с па-

мятью. Однако если какое-нибудь устройство ввода-вывода находится в действии, оно будет запрашивать доступ к шине и получать его каждый раз, когда ему это необходимо. Такой процесс называется **занятием цикла памяти** и замедляет работу компьютера.

Такая система успешно использовалась в первых персональных компьютерах, поскольку все их компоненты работали примерно с одинаковой скоростью. Однако как только центральные процессоры, память и устройства ввода-вывода стали работать быстрее, возникла проблема: шина больше не могла справляться с такой нагрузкой. В случае с закрытыми системами, например рабочими станциями, решением данной проблемы стала разработка новой шины с более высокой скоростью передачи данных для следующей модели машины. Поскольку никто никогда не переносил устройства ввода-вывода со старой модели на новую, такой подход работал успешно.

И все же в мире персональных компьютеров многие заменяли процессор более усовершенствованным, но при этом хотели подсоединить свой старый принтер, сканер и модем к новой системе. Кроме того, существовала целая обширная отрасль промышленности, которая выпускала широкий спектр устройств ввода-вывода для шины IBM PC, и производители этих устройств были совершенно не заинтересованы в том, чтобы начинать все разработки заново. Компания IBM прошла этот тяжелый путь, выпустив после серии IBM PC серию PS/2. У PS/2 была новая шина с более высокой скоростью передачи данных, но большинство производителей клонов продолжали использовать старую шину PC, которая сейчас называется шиной **ISA (Industry Standard Architecture — стандартная промышленная архитектура)**. Большинство производителей дисков и устройств ввода-вывода также продолжали выпускать контроллеры для старой модели, поэтому IBM оказалась в весьма неприятной ситуации, поскольку она в тот момент была единственным производителем персональных компьютеров, несовместимых с серией IBM. В конце концов компания была вынуждена вернуться к производству компьютеров на основе шины ISA. Отметим, что ISA также может быть сокращением от Instruction Set Architecture (архитектура набора команд), если речь идет об уровнях компьютера. А если речь идет о шинах, аббревиатура ISA означает Industry Standard Architecture (стандартная промышленная архитектура).

Тем не менее, несмотря на то, что из-за влияния рынка никаких изменений не произошло, старая шина работала слишком медленно, поэтому что-то нужно было предпринять. Данная ситуация привела к тому, что другие компании начали производить компьютеры с несколькими шинами, одна из которых была старой шиной ISA или **EISA (Extended ISA — расширенная архитектура промышленного стандарта)**. EISA — последователь ISA, совместимый со старыми версиями. В настоящее время самой популярной из них является шина **PCI (Peripheral Component Interconnect — взаимодействие периферийных компонентов)**. Она была разработана компанией Intel, при этом было решено сделать все патенты всеобщим достоянием, чтобы вся компьютерная промышленность (в том числе и конкуренты компании) могла перенять эту идею.

Существует много различных конфигураций шины PCI. Наиболее типичная из них показана на рис. 2.25. В такой конфигурации центральный процессор об-

щается с контроллером памяти по специальному средству связи с высокой скоростью передачи данных. Контроллер соединяется с памятью и шиной PCI непосредственно, и таким образом, передача данных между центральным процессором и памятью происходит не через шину PCI. Однако периферийные устройства с высокой скоростью передачи данных, например SCSI-диски, могут подсоединяться прямо к шине PCI. Кроме того, шина PCI имеет параллельное соединение с шиной ISA, чтобы можно было использовать контроллеры ISA и соответствующие устройства. Машина такого типа обычно содержит 3 или 4 пустых разъема PCI и еще 3 или 4 пустых разъема ISA, чтобы покупатели имели возможность вставлять и старые карты ввода-вывода ISA (для медленно работающих устройств), и новые карты PCI (для устройств с высокой скоростью работы[1]).

В настоящее время существует много разных видов устройств ввода-вывода. Некоторые наиболее распространенные из них описываются ниже.

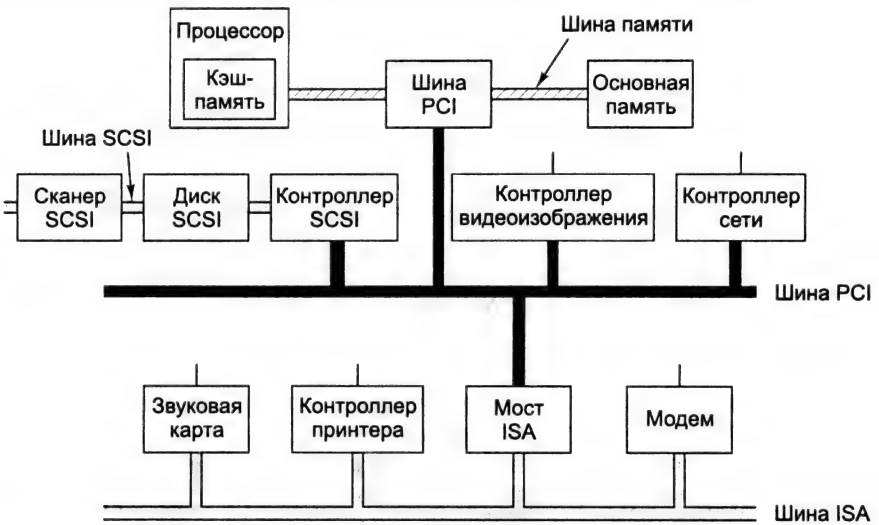


Рис. 2.25. Обычный современный персональный компьютер с шиной PCI и шиной ISA. Модем и звуковая карта — устройства ISA; SCSI-контроллер — устройство PCI

Терминалы

Терминалы компьютера состоят из двух частей: клавиатуры и монитора. В больших компьютерах эти части соединены в одно устройство и связаны с самим компьютером обычным или телефонным проводом. В авиакомпаниях, банках и различных отраслях промышленности, где работают с такими компьютерами, эти устройства до сих пор широко применимы. В мире персональных компьютеров клавиатура и монитор — независимые устройства. Но и в том и в другом случае технология этих двух частей одна и та же.

[1] Необходимо отметить, что в настоящее время существующие стандарты на персональный компьютер уже не содержат шину ISA. — *Примеч. научн. ред.*

Клавиатуры

Существует несколько видов клавиатур. У первых компьютеров IBM PC под каждой клавишей находился переключатель, который давал ощутимую отдачу и щелкал при нажатии клавиши. Сегодня у самых дешевых клавиатур при нажатии клавиш происходит лишь механический контакт с печатной платой. У клавиатур получше между клавишами и печатной платой кладется слой из эластичного материала (особого типа резины). Под каждой клавишей находится небольшой купол, который прогибается в случае нажатия клавиши. Проводящий материал, находящийся внутри купола, замыкает схему. У некоторых клавиатур под каждой клавишей находится магнит, который при нажатии клавиши проходит через катушку и таким образом вызывает электрический ток. Также используются другие методы, как механические, так и электромагнитные.

В персональных компьютерах при нажатии клавиши происходит процедура прерывания и запускается программа обработки прерывания (эта программа является частью операционной системы). Программа обработки прерывания считывает регистр аппаратного обеспечения в контроллер клавиатуры, чтобы получить номер клавиши, которая была нажата (от 1 до 102). Когда клавишу отпускают, происходит второе прерывание. Так, если пользователь нажимает клавишу SHIFT, затем нажимает и отпускает клавишу «M», а затем отпускает клавишу SHIFT, операционная система понимает, что ему нужна заглавная, а не строчная буква «M». Обработка совокупности клавиш SHIFT, CTRL и ALT совершается только программным обеспечением (сюда же относится известное сочетание клавиш CTRL-ALT-DEL, которое используется для перезагрузки всех компьютеров IBM PC и их клонов).

Мониторы с электронно-лучевой трубкой

Монитор представляет собой коробку, содержащую **электронно-лучевую трубку** и ее источники питания. Электронно-лучевая трубка включает в себя электронную пушку, которая выстреливает пучок электронов на фосфоресцентный экран в передней части трубки, как показано на рис. 2.26, *а*. (Цветные мониторы содержат три электронные пушки: одну для красного, вторую для зеленого и третью для синего цвета.) При горизонтальной развертке пучок электронов (луч) развертывается по экрану примерно за 50 мкс, образуя почти горизонтальную полосу на экране. Затем луч совершает горизонтальный обратный ход к левому краю, чтобы начать следующую развертку. Устройство, которое так, линия за линией, создает изображение, называется устройством **растровой развертки**.

Горизонтальная развертка контролируется линейно возрастающим напряжением, которое воздействует на пластины горизонтального отклонения, расположенные слева и справа от электронной пушки. Вертикальная развертка контролируется более медленно возрастающим напряжением, которое воздействует на пластины вертикального отклонения, расположенные под и над электронной пушкой. После определенного количества разверток (от 400 до 1000) напряжение на пластинах вертикального и горизонтального отклонения спадает, и луч возвращается в верхний левый угол экрана. Полное изображение возобновляется от 30 до 60 раз в секунду[1]. Движения луча показаны на рис. 2.26, *б*. Хотя мы описали работу элект-

[1] Современные электронно-лучевые мониторы могут иметь рефреш (частоту обновления изображения, вычерчиваемого лучом на экране) до 150 и более раз в секунду. Эта частота, естественно, обратно пропорционально зависит от количества строк, из которых строится изображение. — *Примеч. научн. ред.*

ронно-лучевых трубок, в которых для развертки луча по экрану используются электрические поля, во многих моделях вместо электрических используются магнитные поля (особенно в дорогостоящих мониторах).

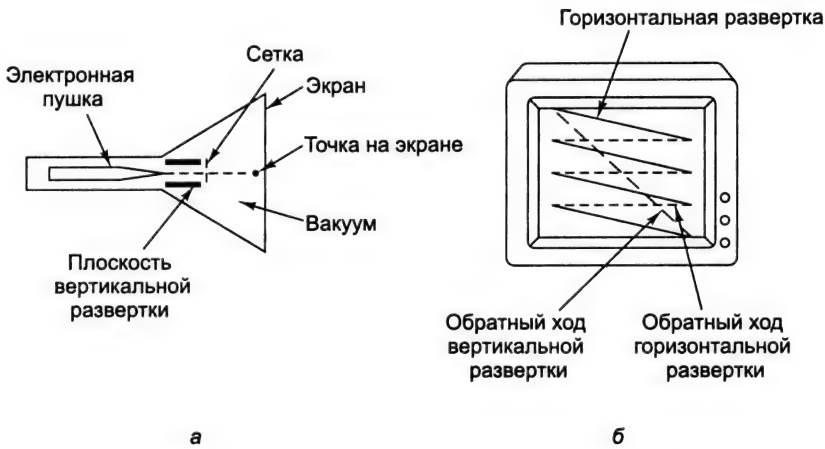


Рис. 2.26. Поперечное сечение электронно-лучевой трубки (*а*); схема развертки электронно-лучевой трубки (*б*)

Для получения на экране изображения из точек внутри электронно-лучевой трубки находится сетка. Когда на сетку воздействует положительное напряжение, электроны возбуждаются, луч направляется на экран, который через некоторое время начинает светиться. Когда используется отрицательное напряжение, электроны отталкиваются и не проходят через сетку, и экран не зажигается. Таким образом напряжение, воздействующее на сетку, вызывает появление соответствующего набора битов на экране. Такой механизм позволяет переводить двоичный электрический сигнал на дисплей, состоящий из ярких и темных точек.

Жидкокристаллические мониторы

Электронно-лучевые трубки слишком громоздкие и тяжелые для использования в портативных компьютерах, поэтому для таких экранов необходима совершенно другая технология. В таких случаях чаще всего используются **жидкокристаллические дисплеи**. Эта технология чрезвычайно сложна, имеет несколько вариантов воплощения и быстро меняется, поэтому мы из необходимости сделаем ее описание по возможности кратким и простым.

Жидкие кристаллы представляют собой вязкие органические молекулы, которые двигаются, как молекулы жидкостей, но при этом имеют структуру, как у кристалла. Они были открыты австрийским ботаником Рейницером (Rheinitzer) в 1888 году и впервые стали применяться при изготовлении различных дисплеев (для калькуляторов, часов и т. п.) в 1960 году. Когда молекулы расположены в одну линию, оптические качества кристалла зависят от направления и поляризации воздействующего света. При использовании электрического поля линия молекул, а следовательно, и оптические свойства могут изменяться. Если воздействовать лучом света на жидкий кристалл, интенсивность света, исходящего из самого жидкого

кристалла, может контролироваться с помощью электричества. Это свойство используется при создании индикаторных дисплеев.

Экран жидкокристаллического дисплея состоит из двух стеклянных параллельно расположенных пластин, между которыми находится герметичное пространство с жидким кристаллом. К обеим пластинам подсоединяются прозрачные электроды. Искусственный или естественный свет за задней пластиной освещает экран изнутри. Электроды, подведенные к пластинам, используются для того, чтобы создать электрические поля в жидком кристалле. На различные части экрана воздействует разное напряжение, и таким образом можно контролировать изображение. К передней и задней пластинам экрана приклеиваются поляроиды, поскольку технология дисплея требует использования поляризованного света. Общая структура показана на рис. 2.27, *а*.

В настоящее время используются различные типы жидкокристаллических дисплеев, но мы рассмотрим только один из них — **дисплей со скрученным нематиком**. В этом дисплее на задней пластине находятся крошечные горизонтальные желобки, а на передней — крошечные вертикальные желобки, как показано на рис. 2.27, *б*. При отсутствии электрического поля молекулы направляются к этим желобкам. Так как они (желобки) расположены перпендикулярно друг к другу, молекулы жидкого кристалла оказываются скрученными на 90°.

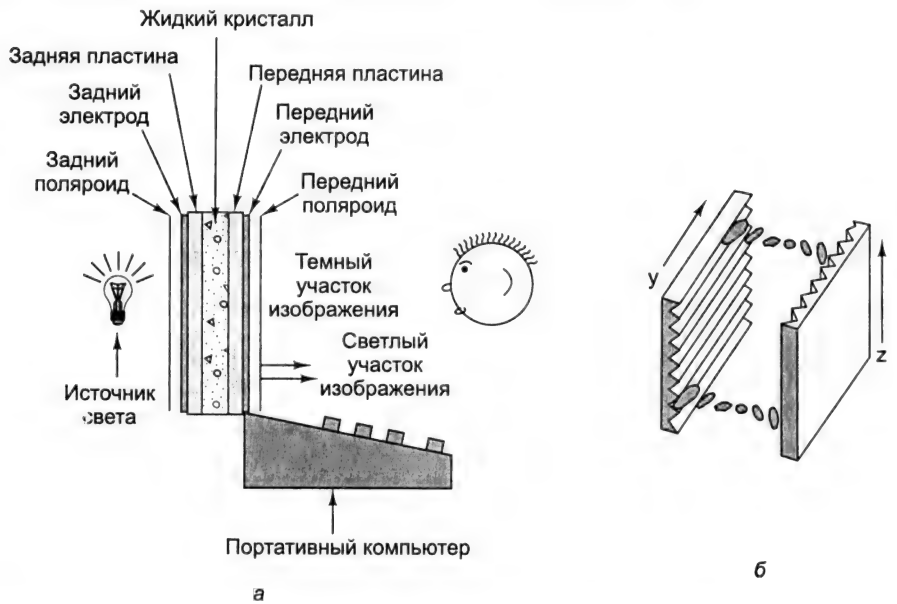


Рис. 2.27. Структура экрана на жидких кристаллах (*а*); желобки на передней и задней пластинах, расположенные перпендикулярно друг к другу (*б*)

На задней пластине дисплея находится горизонтальный поляроид. Он пропускает только горизонтально поляризованный свет. На передней пластине дисплея находится вертикальный поляроид. Он пропускает только вертикально поляризованный свет. Если бы между пластинами не было жидкого кристалла, горизон-

тально поляризованный свет, пропущенный поляроидом на задней пластине, блокировался бы поляроидом на передней пластине, что делало бы экран полностью черным.

Однако скрученная кристаллическая структура молекул, сквозь которую проходит свет, разворачивает плоскость поляризации света. При отсутствии электрического поля жидкокристаллический экран будет полностью освещен. Если подавать напряжение к определенным частям пластины, скрученная структура разрушается, блокируя прохождение света в этих частях.

Для подачи напряжения обычно используются два подхода. В дешевом **пассивном матричном индикаторе** оба электрода содержат параллельные провода. Например, на дисплее размером 640×480 электрод задней пластины содержит 640 вертикальных проводов, а электрод передней пластины — 480 горизонтальных проводов. Если подавать напряжение на один из вертикальных проводов, а затем посылать импульсы на один из горизонтальных, можно изменить напряжение в определенной позиции пиксела и, таким образом, сделать нужную точку темной. Если то же самое повторить со следующим пикселом и т. д., можно получить темную полосу развертки, аналогичную полосам в электронно-лучевых трубках. Обычно изображение на экране перерисовывается 60 раз в секунду, чтобы создавалось впечатление постоянной картинки (так же, как в электронно-лучевых трубках).

Второй подход — применение **активного матричного индикатора**. Он стоит гораздо дороже, чем пассивный матричный индикатор, но зато дает изображение лучшего качества, что является большим преимуществом. Вместо двух наборов перпендикулярно расположенных проводов у активного матричного индикатора имеется крошечный элемент переключения в каждой позиции пиксела на одном из электродов. Меняя состояние переключателей, можно создавать на экране произвольную комбинацию напряжений в зависимости от комбинации битов.

До сих пор мы описывали, как работают монохромные мониторы. Достаточно сказать, что цветные мониторы работают на основе тех же общих принципов, что и монохромные, но детали гораздо сложнее. Чтобы разделить белый цвет на красный, зеленый и синий, в каждой позиции пиксела используются оптические фильтры, поэтому эти цвета могут отображаться независимо друг от друга. Из сочетания этих трех основных цветов можно получить любой цвет.

Символьные терминалы

Обычно используются три типа терминалов: символьные терминалы, графические терминалы и терминалы RS-232-C. Все эти терминалы в качестве входных данных получают набор с клавиатуры, но при этом они отличаются друг от друга тем, каким образом компьютер обменивается с ними информацией, и тем, каким образом передаются выходные данные. Ниже мы кратко опишем каждый из этих типов.

В персональном компьютере существует два способа вывода информации на экран: символьный и графический. На рис. 2.28 показано, как происходит символьное отображение информации на экране (клавиатура считается отдельным устройством). На серийной плате связи находится область памяти, которая называется **видеопамятью**, а также несколько электронных устройств для получения доступа к шине и генерирования видеосигналов.

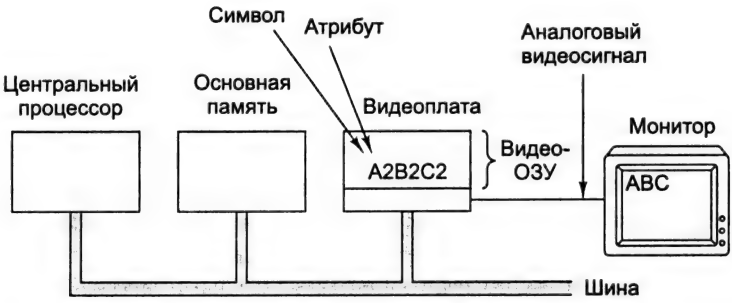


Рис. 2.28. Схема получения выходного сигнала на экране персонального компьютера

Чтобы отобразить на экране символы, центральный процессор копирует их в видеопамять в виде байтов. С каждым символом связывается **атрибутивный байт**, который описывает, какой именно символ должен быть изображен на экране. Атрибуты могут содержать указания на цвет символа, его интенсивность, а также на то, мигает он или нет. Таким образом, изображение 25×80 символов требует наличия 4000 байтов видеопамяти (2000 для символов и 2000 для атрибутов). Большинство плат содержат больше памяти для хранения нескольких изображений.

Видеоплата должна время от времени посылать символы из видео-ОЗУ и порождать необходимый сигнал, чтобы приводить в действие монитор. За один раз посылается целая строка символов, поэтому можно вычислять отдельные строки развертки. Этот сигнал является аналоговым сигналом с высокой частотой, и он контролирует развертку электронного луча, который рисует символы на экране. Так как выходными данными платы является видеосигнал, монитор должен находиться не дальше, чем в нескольких метрах от компьютера, чтобы предотвратить искажение.

Графические терминалы

При втором способе вывода информации на экран видеопамять рассматривается не как массив символов 25×80, а как массив элементов изображения, которые называются **пикселами**. Каждый пиксел может быть включен или выключен. Он представляет один элемент информации. В персональных компьютерах монитор может содержать 640×480 пикселов, но чаще используются мониторы 800×600 и более. Мониторы рабочих станций обычно содержат 1280×960 пикселов и более. Терминалы, отображающие биты, а не символы, называются **графическими терминалами**. Все современные видеоплаты могут работать или как символьные, или как графические терминалы под контролем программного обеспечения.

Основная идея работы терминала показана на рис. 2.28. Однако в случае с графическим изображением видео-ОЗУ рассматривается как большой массив битов. Программное обеспечение может задавать любую комбинацию битов, и она сразу же будет отображаться на экране. Чтобы нарисовать символы, программное обеспечение может, например, назначить для каждого символа прямоугольник 9×14 и заполнять его необходимыми битами. Такой подход позволяет программному обеспечению создавать разнообразные шрифты и сочетать их по желанию. Аппаратное обеспечение только отображает на экране массив битов. Для цветных мониторов каждый пиксел содержит 8, 16 или 24 бита.

Графические терминалы обычно используются для поддержки мониторов, содержащих несколько окон. **Окном** называется область экрана, используемая одной программой. Если одновременно работает несколько программ, на экране появляется несколько окон, при этом каждая программа отображает результаты независимо от других программ.

Хотя графические терминалы универсальны, у них есть два больших недостатка. Во-первых, они требуют большого объема видео-ОЗУ. В настоящее время обычно используются мониторы 640×480 (VGA), 800×600 (SVGA), 1024×768 (XVGA) и 1280×960. Отметим, что у всех этих мониторов отношение ширины и высоты 4:3, что соответствует соотношению сторон телевизионных экранов. Чтобы получить цвет, необходимо 8 битов для каждого из трех основных цветов, или 3 байта на пиксел. Следовательно, для монитора 1024×768 требуется 2,3 Мбайт видео-ОЗУ.

Из-за требования такого большого объема памяти приходится идти на компромисс. При этом для указания цвета используется 8-битный номер. Этот номер является индексом таблицы аппаратного обеспечения, которая называется **цветовой палитрой** и включает в себя 256 разделов, каждый из которых содержит 24 бита. Биты указывают на сочетание красного, зеленого и синего цветов. Такой подход, называемый **индексацией цветов**, сокращает необходимый объем видео-ОЗУ на 2/3, но допускает только 256 цветов. Обычно каждое окно на экране отображается отдельно, но при этом используется только одна цветовая палитра. К тому же, когда на экране присутствуют несколько окон, правильно передаются цвета только одного из них.

Второй недостаток графических терминалов — низкая производительность. Поскольку программисты осознали, что они могут управлять каждым пикселом во времени и пространстве, они, естественно, хотят осуществить эту возможность. Хотя данные могут копироваться из видео-ОЗУ на монитор без прохождения через главную шину, при доставке данных в видео-ОЗУ без использования шины не обойтись. Чтобы отобразить цветное изображение на полный экран размером 1024×768, необходимо копировать 2,3 Мбайт данных в видео-ОЗУ для каждого кадра. Для двигающегося видеоизображения должно сменяться по крайней мере 25 кадров в секунду, а скорость передачи данных должна составлять 57,6 Мбайт/с. Шина (E)ISA не может выдержать такую нагрузку, поэтому необходимо использовать видеокарты PCI, но даже в этом случае приходится идти на компромисс.

Еще одна проблема, связанная с производительностью, — как прокручивать экран. Можно скопировать все биты в программное обеспечение, но это очень сильно перегрузит центральный процессор. Не удивительно, что многие видеокарты оснащены специальным аппаратным обеспечением, которое двигает части экрана не с помощью копирования битов, а путем изменения базовых регистров.

Терминалы RS-232-C

Одни компании производят компьютеры, а другие выпускают терминалы (особенно для больших компьютеров). Чтобы (почти) любой терминал мог работать с (почти) любым компьютером, Ассоциация стандартов в электронной промышленности разработала стандартный интерфейс для терминалов под названием **RS-232-C**.

Терминалы RS-232-C содержат стандартизированный разъем с 25 выводами. Стандарт RS-232-C определяет размер и форму разъема, уровни напряжения и значение сигнала на каждом выводе.

Если компьютер и терминал разделены, чаще всего их можно соединить только по телефонной сети. К несчастью, телефонная сеть не может передавать сигналы, требуемые стандартом RS-232-C, поэтому для преобразования сигнала между компьютером и телефоном, а также между терминалом и телефоном помещается устройство, называемое **модемом** (**мо**дулятор-**дем**одулятор). Ниже мы кратко рассмотрим устройство модемов.

На рис. 2.29 показано расположение компьютера, модемов и терминала при использовании телефонной линии. Если терминал находится достаточно близко от компьютера, так, что их можно связать обычным проводом, модемы не подсоединяются, но в этом случае используются те же кабели и разъемы RS-232-C, хотя выводы, связанные с модемом, не нужны.

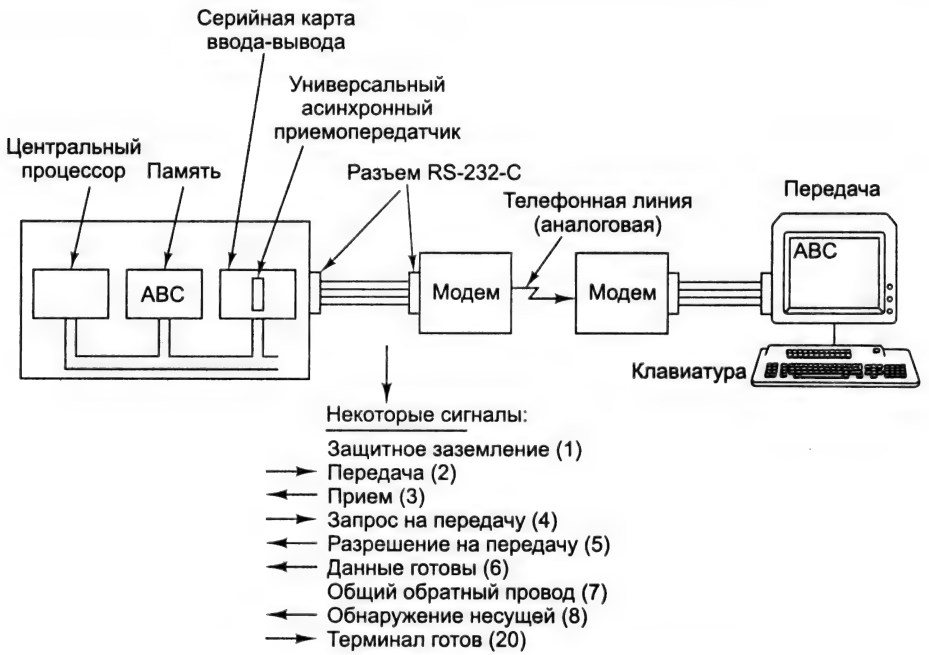


Рис. 2.29. Соединение терминала RS-232-C с компьютером. В списке сигналов в скобках указаны номера выводов

Чтобы обмениваться информацией, и компьютер, и терминал должны содержать микросхему UART (Universal Asynchronous Receiver Transmitter — универсальный асинхронный приемопередатчик), а также логическую схему для доступа к шине. Чтобы отобразить на экране символ, компьютер вызывает этот символ из основной памяти и передает его UART, который затем отправляет его по кабелю RS-232-C бит за битом. В UART поступает сразу целый символ (1 байт), который преобразуется в последовательность битов, и они передаются один за другим с определенной скоростью. UART добавляет к каждому символу начальный и конечный биты, чтобы отделить один символ от другого. При скорости передачи 110 бит/с используется 2 конечных бита.

В терминале другой UART получает биты и восстанавливает целый символ, который затем отображается на экране. Входная информация, которая поступает с клавиатуры терминала, преобразуется в терминале из целых символов в последовательность битов, а затем UART в компьютере восстанавливает целые символы.

Стандарт RS-232-C определяет около 25 сигналов, но на практике используются только некоторые из них (большинство из которых может опускаться, если терминал непосредственно соединен с компьютером проводом, без модемов). Штыри 2 и 3 предназначены для отправки и получения данных соответственно. По каждому выводу проходит односторонний поток битов (один в одном направлении, а другой в противоположном). Когда терминал или компьютер включен, он выдает сигнал готовности терминала (то есть устанавливает 1), чтобы сообщить модему, что он включен. Сходным образом модем выдает сигнал готовности набора данных, чтобы сообщить об их наличии. Когда терминалу или компьютеру нужно послать данные, он выдает сигнал запроса о разрешении пересылки. Если модем разрешает пересылку, он должен выдать сигнал о том, что путь для пересылки свободен. Другие выводы выполняют различные функции определения состояний, проверки и синхронизации.

Мыши

Время идет, а люди работают за компьютером, все меньше и меньше вдаваясь в принципы его работы. Компьютеры серии ENIAC использовались только теми, кто их конструировал. В 50-е годы с компьютерами работали только высококвалифицированные программисты. Сейчас за компьютерами работают многие люди, при этом они не знают (или даже не хотят знать), как работают компьютеры и как они программируются.

Много лет назад у большинства компьютеров был интерфейс с командной строкой, в которой набирались различные команды. Поскольку многие неспециалисты считали такие интерфейсы недружелюбными или даже враждебными, компьютерные фирмы разработали специальные интерфейсы с возможностью указания на экран. Для создания такой возможности чаще всего используется мышь.

Мышь — это маленькая пластиковая коробка, которая лежит на столе рядом с клавиатурой. Если ее двигать по столу, курсор на экране тоже будет двигаться, позволяя пользователям указывать на элементы экрана. У мыши есть одна, две или три кнопки, нажатие на которые дает возможность пользователям выбирать строки меню. Было очень много споров по поводу того, сколько кнопок должно быть у мыши. Наивные пользователи предпочитали одну (так как в этом случае невозможно нажать не ту кнопку), но более продвинутые предпочитали несколько кнопок, чтобы можно было на экране выполнять сложные действия.

Существует три типа мышей: механические, оптические и оптомеханические. У мышей первого типа снизу торчат резиновые колесики, оси которых расположены перпендикулярно друг к другу. Если мышь передвигается в вертикальном направлении, то вращается одно колесо, а если в горизонтальном, то другое. Каждое колесико приводит в действие резистор (потенциометр). Если измерить изменения сопротивления, можно узнать, на сколько провернулось колесико, и таким образом вычислить, на какое расстояние передвинулась мышь в каждом направле-

нии. В последние годы такие мыши были практически полностью вытеснены новой моделью, в которой вместо колес используется шарик, который слегка высовывается снизу. Такая мышь изображена на рис. 2.30.

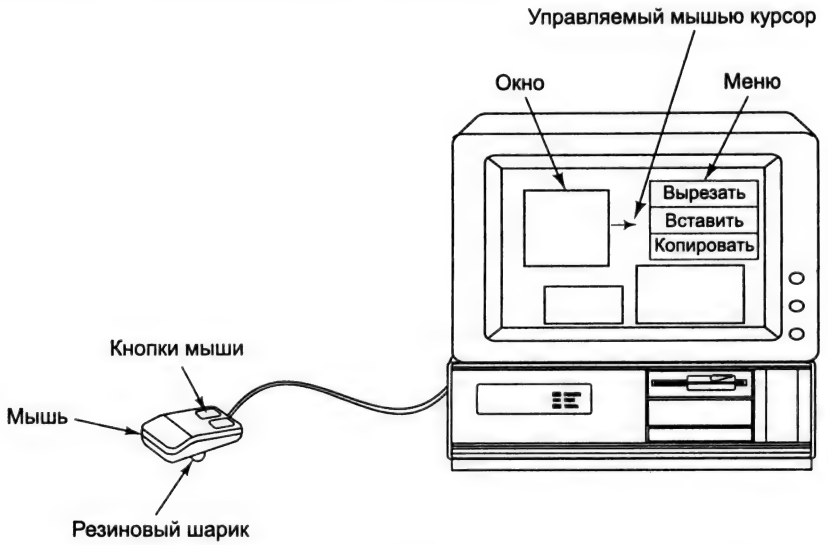


Рис. 2.30. Использование мыши для указания на строки меню

Следующий тип — оптическая мышь. У нее нет ни колес, ни шарика. Вместо этого используются **светодиод** и фотодетектор, расположенный в нижней части мыши. Оптическая мышь перемещается по поверхности особого пластикового коврика, который содержит прямоугольную решетку с линиями, близко расположенными друг к другу. Когда мышь двигается по решетке, фотодетектор воспринимает пересечения линий, наблюдая изменения в количестве света, отражаемого от светодиода. Электронное устройство внутри мыши подсчитывает количество пересеченных линий в каждом направлении.

Третий тип — оптомеханическая мышь. У нее, как и у более современной механической мыши, есть шарик, который вращает два вывода, расположенных перпендикулярно друг к другу. Выводы связаны с кодировщиками. В каждом кодировщике имеются прорези, через которые проходит свет. Когда мышь двигается, выводы вращаются и световые импульсы воздействуют на детекторы каждый раз, когда между светодиодом и детектором появляется прорезь. Число воспринятых детектором импульсов пропорционально количеству перемещения.

Хотя мыши можно устанавливать по-разному, обычно используется следующая система: компьютеру передается последовательность из 3 байтов каждый раз, когда мышь проходит определенное минимальное расстояние (например, 0,01 дюйма). Обычно эти характеристики передаются в последовательном потоке битов. Первый байт содержит целое число, которое указывает, на какое расстояние переместилась мышь в направлении x с прошлого раза. Второй байт содержит ту же информацию для направления y. Третий байт указывает на текущее состояние кнопок мыши. Иногда для каждой координаты используются два байта.

Программное обеспечение принимает эту информацию по мере поступления и преобразует относительные движения, передаваемые мышью, в абсолютную позицию. Затем оно отображает стрелочку на экране в позиции, соответствующей расположению мыши. Если указать стрелочкой на определенный элемент экрана и щелкнуть кнопкой мыши, компьютер может вычислить, какой именно элемент компьютерной информации, соответствующий данному элементу на экране, был выбран.

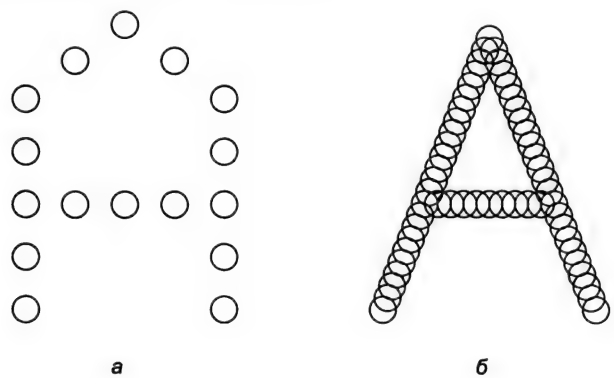
Принтеры

Иногда пользователю нужно напечатать созданный документ или страницу, полученную из World Wide Web, поэтому компьютеры могут быть оснащены принтером. В этом разделе мы опишем некоторые наиболее распространенные типы монохромных (то есть черно-белых) и цветных принтеров.

Монохромные принтеры

Самыми дешевыми являются **матричные принтеры**, у которых печатающая головка последовательно проходит каждую строку печати. Головка содержит от 7 до 24 игл, возбуждаемых электромагнитным полем. Дешевые матричные принтеры имеют 7 игл для печати, скажем, 80 символов в строке в матрице 5×7. В результате строка печати состоит из 7 горизонтальных линий, а каждая из этих линий состоит из 5×80=400 точек. Каждая точка может печататься или не печататься в зависимости от того, какая нужна буква. На рис. 2.31, *а* показана буква «A», напечатанная на матрице 5×7.

Качество печати можно повышать двумя способами: использовать большее количество игл и создавать наложение точек. На рис. 2.31, *б* показана буква «A», напечатанная с использованием 24 игл, в результате чего получилось пересечение точек. Для получения таких пересечений обычно требуется несколько проходов по одной строке печати, поэтому чем выше качество печати, тем медленнее работает принтер. Большинство принтеров можно настраивать, создавая различные варианты соотношения качества и скорости.



а *б*

Рис. 2.31. Буква «A» на матрице 5×7 (*а*); буква «A», напечатанная с использованием 24 игл. Получается наложение точек (*б*)

Матричные принтеры дешевы (особенно в отношении расходных материалов) и очень надежны, но работают медленно, шумно, и качество печати очень низкое. Однако они широко применимы, по крайней мере, в трех областях. Во-первых, они очень популярны для печати на больших листах (более 30 см). Во-вторых, ими очень удобно пользоваться при печати на маленьких отрезках бумаги (например, кассовых чеках, уведомлениях о снятии денег с кредитных карт, посадочных талонах в авиакомпаниях). В-третьих, они используются для распечатывания одновременно нескольких листов с вложенной между ними копировальной бумагой, и эта технология самая дешевая.

Дома удобно использовать недорогие **струйные принтеры**. Подвижная печатающая головка содержит картридж с чернилами. Она двигается горизонтально над бумагой, а чернила в это время выпрыскиваются из крошечных выпускных отверстий. Внутри каждого отверстия капля чернил нагревается до критической точки и в конце концов вырывается наружу. Единственное место, куда она может попасть из отверстия, — лист бумаги. Затем выпускное отверстие охлаждается, в результате создается вакуум, который втягивает следующую каплю. Скорость работы принтера зависит от того, насколько быстро повторяется цикл нагревания/охлаждения. Струйные принтеры обычно имеют разрешающую способность от 300 dpi (dots per inch — точек на дюйм) до 720 dpi, хотя существуют струйные принтеры с разрешающей способностью 1440 dpi. Они достаточно дешево стоят, работают бесшумно и дают хорошее качество печати, однако отличаются низкой скоростью, используют очень дорогие картриджи и производят распечатки, смоченные чернилами.

Вероятно, самым удивительным изобретением в области печатных технологий со времен Йоганна Гутенберга, который изобрел подвижную литеру в XV веке, является **лазерный принтер**. Это устройство сочетает хорошее качество печати, универсальность, высокую скорость работы и умеренную стоимость. В лазерных принтерах используется почти такая же технология, как в фотокопировальных устройствах. Многие компании производят устройства, совмещающие свойства копировальной машины, принтера и иногда также факса.

Основное устройство принтера показано на рис. 2.32. Главной частью этого принтера является вращающийся барабан (в некоторых более дорогостоящих системах вместо барабана используется лента). Перед печатью каждого листа барабан получает напряжение около 1000 вольт и окружается фоточувствительным материалом. Свет лазера проходит вдоль барабана (по длине) почти как пучок электронов в электронно-лучевой трубке, только вместо напряжения для сканирования барабана используется вращающееся восьмиугольное зеркало. Луч света модулируется, и в результате получается определенный набор темных и светлых участков. Участки, на которые воздействует луч, теряют свой электрический заряд.

После того как нарисована строка точек, барабан немного поворачивается для создания следующей строки. В итоге первая строка точек достигает резервуара с тонером (электростатически чувствительным черным порошком). Тонер притягивается к тем точкам, которые заряжены, и так формируется визуальное изображение строки. Через некоторое время барабан с тонером прижимается к бумаге, оставляя на ней отпечаток изображения. Затем лист проходит через горячие валики, и изображение закрепляется. После этого барабан разряжается, и остатки тонера счищаются с него. Теперь он готов к печатанию следующей страницы.

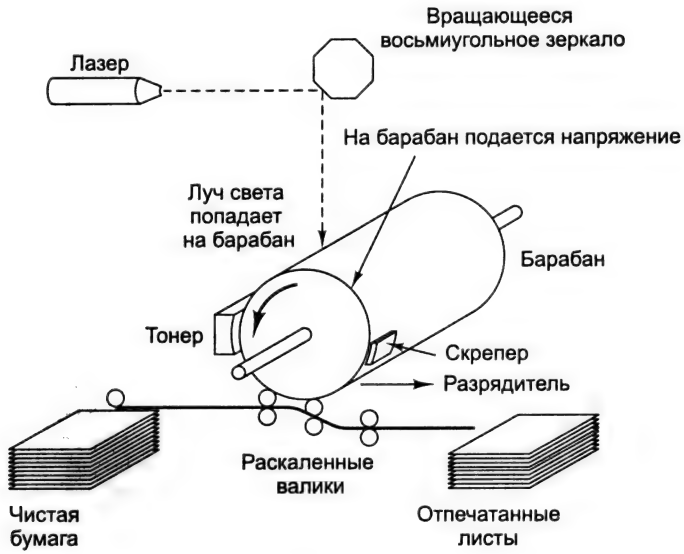


Рис. 2.32. Работа лазерного принтера

Едва ли нужно говорить, что этот процесс представляет собой чрезвычайно сложную комбинацию физики, химии, механики и оптики. Тем не менее некоторые производители выпускают агрегаты, которые называются **печатающими устройствами**. Изготовители лазерных принтеров сочетают печатающие устройства с их собственным аппаратным и программным обеспечением. Аппаратное обеспечение состоит из быстроработающего процессора, а также нескольких мегабайтов памяти для хранения полного изображения в битовой форме и различных шрифтов, одни из которых встроены, а другие загружаются из памяти. Большинство принтеров принимают команды, описывающие страницу, которую нужно напечатать (в противоположность принтерам, принимающим изображения в битовой форме от центрального процессора). Эти команды обычно даются на языке PCL или PostScript.

Лазерные принтеры с разрешающей способностью 300 dpi и выше могут печатать черно-белые фотографии, но технология при этом гораздо сложнее, чем может показаться на первый взгляд. Рассмотрим фотографию, отсканированную с разрешающей способностью 600 dpi, которую нужно напечатать на принтере с такой же разрешающей способностью (600 dpi). Сканированное изображение содержит 600×600 пикселов/дюйм, каждый пиксел характеризуется определенной степенью серого цвета от 0 (белый цвет) до 255 (черный цвет). Принтер может печатать с разрешающей способностью 600 dpi, но каждый напечатанный пиксел может быть либо черного цвета (когда есть тонер), либо белого цвета (когда нет тонера). Степени серого печататься не могут.

Для печати таких изображений используется так называемая **обработка полутонов** (как при печати серийных плакатов). Изображение разбивается на ячейки, каждая 6×6 пикселов. Каждая ячейка может содержать от 0 до 36 черных пикселов. Человеческому глазу ячейка с большим количеством черных пикселов кажется

темнее, чем ячейка с небольшим количеством черных пикселов. Серые тона в диапазоне от 0 до 255 передаются следующим образом. Этот диапазон делится на 37 зон. Серые тона от 0 до 6 расположены в зоне 0, от 7 до 13 — в зоне 1 и т. д. (зона 36 немного меньше, чем другие, потому что 256 на 37 без остатка не делится). Когда встречаются тона зоны 0, ячейка оставляется белой, как показано на рис. 2.33, *а*. Тона зоны 1 передаются одним черным пикселом в ячейке. Тона зоны 2 — двумя пикселами в ячейке, как показано на рис. 2.33, *б*. Изображение серых тонов других зон показано на рис. 2.33, *в* — *е*. Если фотография отсканирована с разрешающей способностью 600 dpi, после подобной обработки полутонов разрешающая способность напечатанного изображения снижается до 100 ячеек/дюйм. Данная разрешающая способность называется **градацией полутонов** и измеряется в **lpi** (lines per inch — количество строк на дюйм).

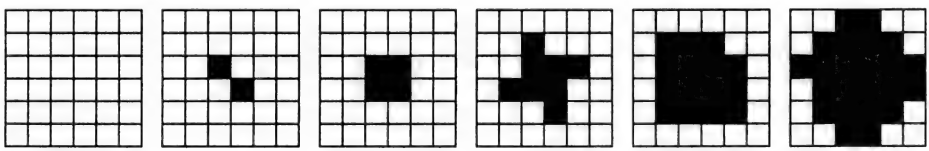


Рис. 2.33. Изображение серых полутонов различных зон: 0–6 (*а*); 14–20 (*б*); 28–34 (*в*); 56–62 (*г*); 105–111 (*д*); 161–167 (*е*)

Цветные принтеры

Цветные изображения могут передаваться двумя способами: с помощью поглощенного света и с помощью отраженного света. Поглощенный свет используется, например, при создании изображений в электронно-лучевых мониторах. В данном случае изображение строится путем аддитивного наложения трех основных цветов: красного, зеленого и синего. Отраженный свет используется при создании цветных фотографий и картинок в глянцевых журналах. В этом случае поглощается свет с определенной длиной волны, а остальной свет отражается. Такие изображения создаются путем субтрактивного наложения трех основных цветов: голубого (красный полностью поглощен), желтого (синий полностью поглощен) и сиреневого (зеленый полностью поглощен). Теоретически путем смешивания голубых, желтых и сиреневых чернил можно получить любой цвет. Но на практике очень сложно получить такие чернила, которые полностью поглощали бы весь свет и в результате давали черный цвет. По этой причине практически во всех цветных печатающих устройствах используются чернила четырех цветов: голубого, желтого, сиреневого и черного. Такая система цветов называется CYMK (C — Cyan (голубой), Y — Yellow (желтый) M — Magenta (сиреневый) и K — blacK (черный)). Из слова «black» берется последняя буква, чтобы не путать с Blue (синий). Мониторы, напротив, используют поглощенный свет и наложение красного, зеленого и синего цветов для создания цветного изображения.

Полный набор цветов, который может производить монитор или принтер, называется **цветовой шкалой**. Не существует такого устройства, которое полностью передавало бы цвета окружающего нас мира. В лучшем случае устройство дает всего

256 степеней интенсивности каждого цвета, и в итоге получается только 16 777 216 различных цветов. Несовершенство технологий еще больше сокращает это число, а оставшиеся цвета не передают полного цветового спектра. Кроме того, цветовосприятие связано не только с физическими свойствами света, но и с работой палочек и колбочек в сетчатке глаза.

Из всего этого следует, что превратить красивое цветное изображение, которое замечательно смотрится на экране, в идентичное печатное изображение очень сложно. Среди основных проблем можно назвать следующие:

1. Цветные мониторы используют поглощенный свет; цветные принтеры используют отраженный свет.
2. Электронно-лучевая трубка производит 256 оттенков каждого цвета; цветные принтеры должны совершать обработку полутонов.
3. Мониторы имеют темный фон; бумага имеет светлый фон.
4. Системы цветов RGB (Red, Green, Blue — красный, зеленый, синий) и CYMK отличаются друг от друга.

Чтобы цветные печатные изображения соответствовали реальной действительности (или хотя бы изображениям на экране), необходима калибровка оборудования, сложное программное обеспечение и компетентность пользователя.

Для цветной печати используются пять технологий, и все они основаны на системе CYMK. Самыми дешевыми являются **цветные струйные принтеры**. Они работают так же, как и монохромные струйные принтеры, но вместо одного картриджа в них находится четыре (для голубых, желтых, сиреневых и черных чернил). Они хорошо печатают цветную графику, сносно печатают фотографии и при этом не очень дорого стоят (отметим, что сами принтеры дешевые, а картриджи довольно дорогие).

Для получения лучших результатов должны использоваться особые чернила и особая бумага. Существует два вида чернил. **Чернила на основе красителя** состоят из красителей, растворенных в жидкой среде. Они дают яркие цвета и легко вытекают из картриджа. Главным недостатком таких чернил является то, что они быстро выгорают под воздействием ультрафиолетовых лучей, которые содержатся в солнечном свете. **Чернила на основе пигмента** содержат твердые частицы пигмента, погруженные в жидкость. Жидкость испаряется с бумаги, а пигмент остается. Чернила не выгорают, но зато дают не такие яркие краски, как чернила на основе красителя. Кроме того, частицы пигмента часто засоряют выпускные отверстия картриджей, поэтому их нужно периодически чистить. Для печати фотографий необходима мелованная или глянцевая бумага. Эти особые виды бумаги были созданы специально для того, чтобы удерживать капельки чернил и не давать им растекаться.

Следующий тип принтеров — **принтеры с твердыми чернилами**. В этих принтерах содержится 4 твердых блока специальных восковых чернил, которые затем расплавляются. Перед началом печати должно пройти 10 минут (время, необходимое для того, чтобы расплавить чернила). Горячие чернила выпрыскиваются на бумагу, где они затвердевают и закрепляются после прохождения листа между двумя валиками.

Третий тип цветных принтеров — **цветные лазерные принтеры**. Они работают так же, как их монохромные братья, только они составляют четыре отдельных изображения (голубого, желтого, сиреневого и черного цвета) и используют четыре разных тонера. Поскольку полное изображение в битовой форме обычно составляется заранее, для изображения с разрешающей способностью 1200×1200 dpi на листе в 80 квадратных дюймов нужно 115 млн пикселов. Так как каждый пиксел состоит из 4 битов, принтеру нужно 55 Мбайт памяти только для хранения изображения в битовой форме, не считая памяти, необходимой для внутренних процессоров, шрифтов и т. п. Это требование делает цветные лазерные принтеры очень дорогими, но зато они очень быстро работают и дают высокое качество печати. К тому же полученные изображения сохраняются на протяжении длительного времени.

Четвертый тип принтеров — **принтеры с восковыми чернилами**. Они содержат широкую ленту из четырехцветного воска, которая разделяется на отрезки размером с лист бумаги. Тысячи нагревательных элементов растапливают воск, когда бумага проходит под лентой. Воск закрепляется на бумаге в форме пикселов с использованием системы CYMK. Такие принтеры когда-то были очень популярны, но сейчас их вытеснили другие типы принтеров с более дешевыми расходными материалами.

Пятый тип принтеров работает на основе технологии **сублимации**. Это слово содержит некоторые фрейдистские нотки[1], однако в науке под сублимацией понимается переход твердых веществ в газообразные без прохождения через стадию жидкости. Таким материалом является, например, сухой лед (замороженный углекислый газ). В принтере, работающем на основе процесса сублимации, контейнер с красителями CYMK двигается над термической печатающей головкой, которая содержит тысячи программируемых нагревательных элементов. Красители мгновенно испаряются и впитываются специальной бумагой. Каждый нагревательный элемент может производить 256 различных температур. Чем выше температура, тем больше красителя осаждается и тем интенсивнее получается цвет. В отличие от всех других цветных принтеров, данный принтер способен воспроизводить цвета практически сплошного спектра, поэтому процедура обработки полутонов не нужна. Процесс сублимации часто используется при изготовлении моментальных снимков. Такие снимки делаются на специальной дорогостоящей бумаге.

Модемы

С появлением большого количества компьютеров в последние годы возникла необходимость установить связь между компьютерами. Например, можно связать свой домашний компьютер с компьютером на работе, с поставщиком услуг Интернета или банковской системой. Для обеспечения такой связи часто используется телефонная линия.

Однако грубая телефонная линия не подходит для передачи компьютерных сигналов, которые обычно передают 0 как 0 В, а 1 — от 3 до 5 В, как показано на рис. 2.34, *а*. Двухуровневые сигналы страдают от сильного искажения во время

[1] Сублимация в психологии означает психический процесс преобразования и переключения энергии влечений на цели социальной деятельности и культурного творчества; термин введен З. Фрейдом. — *Примеч. перев.*

передачи по телефонной линии, которая предназначена для передачи голоса, а искажения ведут к ошибкам в передаче. Тем не менее синусоидальный сигнал с частотой от 1000 до 2000 Гц, который называется **несущим сигналом**, может передаваться с относительно небольшими искажениями, и это свойство используется при передаче данных в большинстве телекоммуникационных систем.

Поскольку синусоидальная волна полностью предсказуема, она не передает никакой информации. Однако изменяя амплитуду, частоту или фазу, можно передавать последовательность нулей и единиц, как показано на рис. 2.34. Этот процесс называется **модуляцией**. При **амплитудной модуляции** (рис. 2.34, *б*) используется два уровня напряжения, для 0 и 1 соответственно. Если цифровые данные передаются с очень низкой скоростью, то при передаче 1 слышен громкий шум, а при передаче 0 шум отсутствует.

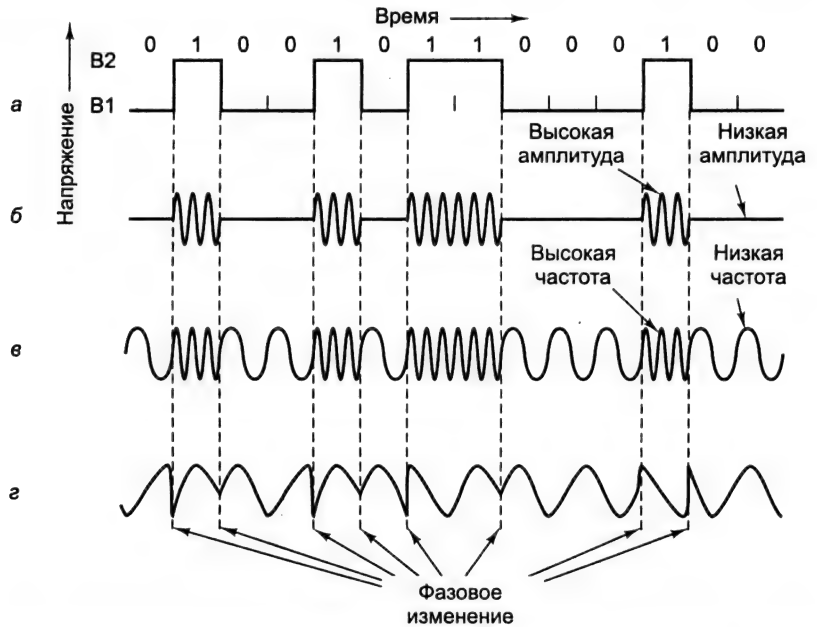


Рис. 2.34. Последовательная передача двоичного числа 01001011000100 по телефонной линии: двухуровневый сигнал (*а*); амплитудная модуляция (*б*); частотная модуляция (*в*); фазовая модуляция (*г*)

При **частотной модуляции** (рис. 2.34, *в*) уровень напряжения не изменяется, но частота несущего сигнала различается для 1 и для 0. В этом случае при передаче цифровых данных можно услышать два тона: один из них соответствует 0, а другой — 1. Частотная модуляция иногда называется **частотной манипуляцией**.

При простой фазовой модуляции (рис. 2.34, *г*) амплитуда и частота сохраняются на одном уровне, а фаза несущего сигнала изменяется на 180 градусов, когда данные меняются с 0 на 1 или с 1 на 0. В более сложных системах фазовой модуляции в начале каждого неделимого временного отрезка фаза несущего сигнала резко сдвигается на 45, 135, 225 или 315 градусов, чтобы передавать 2 бита за один временной отрезок. Это называется **дибитной фазовой кодировкой**. Например,

сдвиг по фазе на 45° представляет 00, сдвиг по фазе на 135° — 01 и т. д. Существуют системы для передачи трех и более битов за один временной отрезок. Число таких временных интервалов (то есть число потенциальных изменений сигнала в секунду) называется скоростью в **бодах**. При передаче двух или более битов за 1 временной отрезок скорость передачи битов будет превышать скорость в **бодах**. Отметим, что термины «бод» и «бит» часто путают.

Если данные состоят из последовательности 8-битных символов, было бы желательно иметь средство связи для передачи 8 битов одновременно, то есть 8 пар проводов. Так как телефонные линии, предназначенные для передачи голоса, обеспечивают только один канал связи, биты должны пересылаться последовательно один за другим (или в группах по два, если используется дибитная кодировка). Устройство, которое получает символы из компьютера в форме двухуровневых сигналов (по одному биту в отрезок времени) и передает биты по одному или по два в форме амплитудной, фазовой или частотной модуляции, называется модемом. Для указания на начало и конец каждого символа в начале и конце 8-битной цепочки ставятся начальный и конечный биты, таким образом, всего получается 10 битов.

Модем посылает отдельные биты каждого символа через равные временные отрезки. Например, скорость 9600 бод означает, что сигнал меняется каждые 104 мкс. Второй модем, получающий информацию, преобразует модулированный несущий сигнал в двоичное число. Биты поступают в модем через равные промежутки времени. Если модем определил начало символа, его часы сообщают, когда нужно начать считывать поступающие биты.

Современные модемы передают данные со скоростью от 28 800 бит/с до 57 600 бит/с, что обычно соответствует более низкой скорости в бодах. Они сочетают несколько технологий для передачи нескольких битов за 1 бод, модулируя амплитуду, частоту и фазу. Почти все современные модемы являются **дуплексными**, то есть могут передавать информацию в обоих направлениях одновременно, используя различные частоты. Модемы и линии связи, которые не могут передавать информацию в обоих направлениях одновременно (как железная дорога, по которой поезда могут ходить и в северном, и в южном направлениях, но не в одно и то же время), называются **полудуплексными**. Линии связи, которые могут передавать информацию только в одном направлении, называются **симплексными**.

ISDN

В начале 80-х годов европейские компании почтовой, телефонной и телеграфной связи разработали стандарт цифровой телефонии под названием **ISDN (Integrated Services Digital Network — цифровая сеть с предоставлением комплексных услуг)**. Она давала возможность горожанам иметь дома сигнализацию, связанную со специальными учреждениями, а также предназначалась для выполнения других своеобразных функций. Компании настойчиво рекламировали эту идею, но без особого успеха. Вдруг появился World Wide Web, и людям понадобился цифровой доступ к Интернету. Тут-то и обнаружилось совершенно потрясающее применение ISDN (хотя вовсе не благодаря разработчикам этой сети). С тех пор она стала очень популярной в США и других странах.

Когда клиент телефонной компании подписывается на ISDN, телефонная компания заменяет старую аналоговую линию новой цифровой. (В действительности сама линия не меняется, меняется только оборудование на обоих концах.) Новая

линия содержит два независимых цифровых канала, каждый со скоростью передачи данных 64 000 бит/с, плюс канал для сигналов со скоростью передачи 16 000 бит/с. Оборудование необходимо для того, чтобы объединить все три канала в один цифровой канал со скоростью передачи данных 144 000 бит/с. Предприятия могут приобретать 30-канальную линию ISDN.

ISDN не только быстрее передает данные, чем аналоговый канал, но и быстрее устанавливает соединение (не дольше 1 секунды), не требует наличия аналогового модема, а также более надежен, то есть дает меньше ошибок, чем аналоговый канал. Кроме того, ISDN имеет ряд дополнительных особенностей, которых нет у аналоговых каналов.

Структура связи ISDN показана на рис. 2.35. Поставщик предоставляет цифровой канал, который передает биты. Что означают эти биты — личное дело отправителя и получателя. Между оборудованием клиента и поставщика помещается устройство для взаимной связи NT1 с T-интерфейсом на одной стороне и U-интерфейсом на другой. В США клиенты должны покупать собственное устройство NT1, а во многих европейских странах — брать напрокат у поставщика.

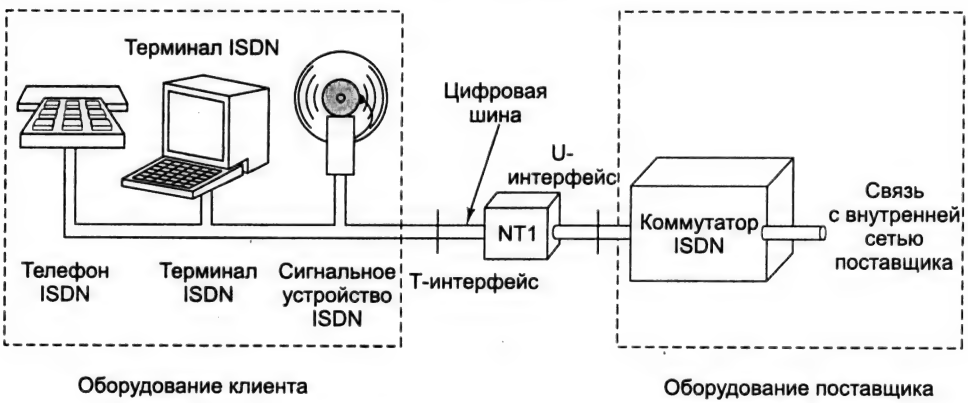


Рис. 2.35. ISDN для домашнего использования

Коды символов

У каждого компьютера есть набор символов, который он использует. Как минимум этот набор включает 26 заглавных и 26 строчных букв[1], цифры от 0 до 9, а также некоторые специальные символы: пробел, точка, запятая, минус, символ возврата каретки и т. д.

Для того чтобы передавать эти символы в компьютер, каждому из них приписывается номер: например, a=1, b=2, ..., z=26, +=27, −=28. Отображение символов в целые числа называется **кодом символов**. Важно отметить, что связанные между собой компьютеры должны иметь один и тот же код, иначе они не смогут обмениваться информацией. По этой причине были разработаны стандарты. Ниже мы рассмотрим два самых важных из них.

[1] Для английского языка. — *Примеч. перев.*

ASCII

Один широко распространенный код называется **ASCII (American Standard Code for Information Interchange — американский стандартный код для обмена информацией)**. Каждый символ ASCII-кода содержит 7 битов, таким образом, всего может быть 128 символов (табл. 2.5). Коды от 0 до 1F (в шестнадцатеричной системе счисления) соответствуют управляющим символам, которые не печатаются.

Многие непечатные символы ASCII предназначены для передачи данных. Например, послание может состоять из символа начала заголовка SOH (Start of Header), самого заголовка, символа начала текста STX (Start of Text), самого текста, символа конца текста ETX (End of Text) и, наконец, символа конца передачи EOT (End of Transmission). Однако на практике послания, отправляемые по телефонным линиям и сетям, форматируются по-другому, так что непечатные символы передачи ASCII практически не используются.

Печатные символы ASCII наглядны. Они включают буквы верхнего и нижнего регистров, цифры, знаки пунктуации и некоторые математические символы.

Таблица 2.5. Таблица кодов ASCII

Число	Команда	Значение	Число	Команда	Значение
0	NUL	Null (Пустой указатель)	10	DLE	Data Link Escape (Выход из системы передачи)
1	SOH	Start of Heading (Начало заголовка)	11	DC1	Device Control 1 (Управление устройством)
2	STX	Start of Text (Начало текста)	12	DC2	Device Control 2 (Управление устройством)
3	ETX	End of Text (Конец текста)	13	DC3	Device Control 3 (Управление устройством)
4	EOT	End of Transmission (Конец передачи)	14	DC4	Device Control 4 (Управление устройством)
5	ENQ	ENQurity (Запрос)	15	NAK	Negative AcKnolidgement (Неподтверждение приема)
6	ACK	ACKnoligement (Подтверждение приема)	16	SYN	SYNcronous idle (Простой)
7	BEL	Bell (Символ звонка)	17	ETB	End of Transmission Block (Конец блока передачи)
8	BS	BackSpace (Отступ назад)	18	CAN	CANcel (Отмена)
9	HT	Horizontal Tab (Горизонтальная табуляция)	19	EM	End of Medium (Конец носителя)
A	LF	Line Feed (Перевод строки)	1A	SUB	SUBstitute (Подстрочный индекс)
B	VT	Vertical Tab (Вертикальная табуляция)	1B	ESC	ESCape (Выход)

Число	Команда	Значение	Число	Команда	Значение
C	FF	From Feed (Перевод страницы)	1C	FS	File Separator (Разделитель файлов)
D	CR	Carriage Return (Возврат каретки)	1D	GS	Group Separator (Разделитель группы)
E	SO	Shift Out (Переключение на дополнительный регистр)	1E	RS	Record Separator (Разделитель записи)
F	SI	Shift In (Переключение на стандартный регистр)	1F	US	Unit Separator (Разделитель модуля)

Число	Символ	Число	Символ	Число	Символ	Число	Символ	Число	Символ	Число	Символ
20	(пробел)	30	0	40	@	50	P	60	‘	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	“	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

UNICODE

Компьютерная промышленность развивалась преимущественно в США, что привело к появлению кода ASCII. Этот код подходит для английского языка, но не очень удобен для других языков. Во французском языке есть надстрочные знаки (например, système), в немецком — умляуты (например, für) и т. д. В некоторых европейских языках есть несколько букв, которых нет в ASCII, например, немецкое β или датское ø. Некоторые языки имеют совершенно другой алфавит (например, русский или арабский), а у некоторых вообще нет алфавита (например, китайский). Компьютеры распространились по всему свету, и поставщики программного обеспечения хотят реализовывать свою продукцию не только в англоязычных, но и в тех странах, где большинство пользователей не говорят по-английски и где нужен другой набор символов.

Первой попыткой расширения ASCII был IS 646, который добавлял к ASCII еще 128 символов, в результате чего получился 8-битный код под названием **Latin-1**. Добавлены были в основном латинские буквы со штрихами и диакритическими знаками. Следующей попыткой был IS 8859, который ввел понятие **кодовая страница**. Кодовая страница — набор из 256 символов для определенного языка или группы языков. IS 8859-1 — это Latin-1. IS 8859-2 включает славянские языки с латинским алфавитом (например, чешский, польский и венгерский). IS 8859-3 содержит символы турецкого, мальтийского, эсперанто и галисийского языков и т. д. Главным недостатком такого подхода является то, что программное обеспечение должно следить, с какой именно кодовой страницей оно имеет дело в данный момент, и при этом невозможно смешивать языки. К тому же эта система не охватывает японский и китайский языки.

Группа компьютерных компаний разрешила эту проблему, создав новую систему под названием UNICODE, и объявила эту систему международным стандартом (IS 10646). UNICODE поддерживается некоторыми языками программирования (например, Java), некоторыми операционными системами (например, Windows NT) и многими приложениями. Вероятно, эта система будет распространяться по всему миру.

Основная идея UNICODE — приписывать каждому символу единственное постоянное 16-битное значение, которое называется **указателем кода**. Многобайтные символы и escape-последовательности не используются. Поскольку каждый символ состоит из 16 битов, писать программное обеспечение гораздо проще.

Так как символы UNICODE состоят из 16 битов, всего получается 65 536 кодовых указателей. Поскольку во всех языках мира в общей сложности около 200 000 символов, кодовые указатели являются очень скудным ресурсом, который нужно распределять с большой осторожностью. Около половины кодов уже распределено, и консорциум, разработавший UNICODE, постоянно рассматривает предложения на распределение оставшейся части. Чтобы ускорить принятие UNICODE, консорциум использовал Latin-1 в качестве кодов от 0 до 255, легко преобразуя ASCII в UNICODE.

Во избежание излишней растраты кодов каждый диакритический знак имеет свой собственный код. А сочетание диакритических знаков с буквами — задача программного обеспечения.

Вся совокупность кодов разделена на блоки, каждый блок содержит 16 кодов. Каждый алфавит в UNICODE имеет ряд последовательных зон. Приведем некоторые примеры (в скобках указано число задействованных кодов): латынь (336), греческий (144), русский (256), армянский (96), иврит (112), деванагари (128), гурмуки (128), ория (128), телугу (128) и каннада (128). Отметим, что каждому из этих языков приписано больше кодов, чем в нем есть букв. Это было сделано отчасти потому, что во многих языках у каждой буквы есть несколько вариантов. Например, каждая буква в английском языке представлена в двух вариантах: там есть строчные и заглавные буквы. В некоторых языках буквы имеют три или более форм, выбор которых зависит от того, где находится буква: в начале, конце или середине слова.

Кроме того, некоторые коды были приписаны диакритическим знакам (112), знакам пунктуации (112), подстрочным и надстрочным знакам (48), знакам валют (48), математическим символам (256), геометрическим фигурам (96) и рисункам (192).

Затем идут символы для китайского, японского и корейского языков. Сначала идут 1024 фонетических символа (например, катакана и бопомофо), затем иероглифы, используемые в китайском и японском языках (20 992), а затем слоги корейского языка (11 156).

Чтобы пользователи могли создавать новые символы для особых целей, существует еще 6400 кодов.

Хотя UNICODE разрешил многие проблемы, связанные с интернационализацией, он все же не мог разрешить абсолютно все проблемы. Например, латинский алфавит упорядочен, а иероглифы — нет, поэтому программа для английского языка может расположить слова «cat» и «dog» по алфавиту, сравнив значение кодов первых букв, а программе для японского языка нужны дополнительные таблицы, чтобы можно было вычислять, в каком порядке расположены символы в словаре.

Еще одна проблема состоит в том, что постоянно появляются новые слова. 50 лет назад никто не говорил об апплетах, киберпространстве, гигабайтах, лазерах, модемах, «смайликах» или видеопленках. С появлением новых слов в английском языке новые коды не нужны. А вот в японском нужны. Кроме новых терминов, необходимо также добавить по крайней мере 20 000 новых имен собственных и географических названий (в основном китайских). Шрифт Брайля, которым пользуются слепые, вероятно, тоже должен быть задействован. Представители различных профессиональных кругов также заинтересованы в наличии каких-либо особых символов. Консорциум по созданию UNICODE рассматривает все новые предложения и выносит по ним решения.

UNICODE использует один и тот же код для символов, которые выглядят почти одинаково, но имеют несколько значений или пишутся немного по-разному в китайском и японском языках (как если бы английские текстовые процессоры всегда писали слово «blue» как «blew», потому что они произносятся одинаково). Одни считают такой подход оптимальным для экономии скудного запаса кодов, другие рассматривают его как англо-саксонский культурный империализм (а вы думали, что приписывание символам 16-битных значений не носит политического характера?). Дело усложняется тем, что полный японский словарь содержит 50 000 иероглифических знаков (не считая собственных имен), поэтому при наличии 20 992 кодов приходится делать выбор и чем-то жертвовать. Далеко не все японцы считают, что консорциум компьютерных компаний, даже если некоторые из них японские, является идеальным форумом, чтобы принимать решения, чем именно нужно жертвовать.

Краткое содержание главы

Компьютерные системы состоят из трех типов компонентов: процессоров, памяти и устройств ввода-вывода. Задача процессора заключается в том, чтобы последовательно вызывать команды из памяти, декодировать и выполнять их. Цикл вызов—декодирование—выполнение всегда можно представить в виде алгоритма. Вызов, декодирование и выполнение команд определенной программы иногда выполняются программой-интерпретатором, работающей на более низком уровне. Для повышения скорости работы во многих компьютерах имеется один или не-

сколько конвейеров или суперскалярная архитектура с нескольким функциональными блоками, которые действуют параллельно.

Широко распространены системы с несколькими процессорами. Компьютеры с параллельной обработкой включают векторные процессоры, в которых одна и та же операция выполняется одновременно над разными наборами данных, мультипроцессоры, в которых несколько процессоров разделяют общую память, и мультикомпьютеры, в которых у каждого компьютера есть своя собственная память, но при этом компьютеры связаны между собой и пересылают друг другу сообщения.

Память можно разделить на основную и вспомогательную. Основная память используется для хранения программ, которые выполняются в данный момент. Время доступа невелико (максимум несколько десятков наносекунд) и не зависит от адреса, к которому происходит обращение. Кэш-память еще больше сокращает время доступа. Память может быть оснащена кодом с исправлением ошибок для повышения надежности.

Время доступа к вспомогательной памяти, напротив, гораздо больше (от нескольких миллисекунд и более) и зависит от расположения считываемых и записываемых данных. Наиболее распространенные виды вспомогательной памяти — магнитные ленты, магнитные диски и оптические диски. Магнитные диски существуют в нескольких вариантах: дискеты, винчестеры, IDE-диски, SCSI-диски и RAID-массивы. Среди оптических дисков можно назвать компакт-диски, диски CD-R и DVD.

Устройства ввода-вывода используются для передачи информации в компьютер и из компьютера. Они связаны с процессором и памятью одной или несколькими шинами. В качестве примеров можно назвать терминалы, мыши, принтеры и модемы. Большинство устройств ввода-вывода используют код ASCII, хотя UNICODE уже стремительно распространяется по всему миру.

Вопросы и задания

1. Рассмотрим машину с трактом данных, который изображен на рис. 2.2. Предположим, что загрузка регистров АЛУ занимает 5 нс, работа АЛУ — 10 нс, а помещение результата обратно в регистр — 5 нс. Какое максимальное число миллионов команд в секунду способна выполнять эта машина при отсутствии конвейера?
2. Зачем нужен шаг 2 в списке шагов, приведенном в разделе «Выполнение команд»? Что произойдет, если этот шаг пропустить?
3. На компьютере 1 выполнение каждой команды занимает 10 нс, а на компьютере 2 – 5 нс. Можете ли вы с уверенностью сказать, что компьютер 2 работает быстрее? Аргументируйте ответ.
4. Предположим, что вы разрабатываете компьютер на одной микросхеме для использования во встроенных системах. Вся память находится на микросхеме и работает с той же скоростью, что и центральный процессор. Рассмотрите принципы, изложенные в разделе «Принципы разработки современных компьютеров», и скажите, важны ли они в данном случае (высокая производительность желательна).

5. Можно ли добавить кэш-память к процессорам, изображенным на рис. 2.7, *б*? Если можно, то какую проблему нужно будет решить в первую очередь?
6. В некотором вычислении каждый последующий шаг зависит от предыдущего. Что в данном случае более уместно: векторный процессор или конвейер? Объясните, почему.
7. Чтобы конкурировать с недавно изобретенным печатным станком, один средневековый монастырь решил наладить массовое производство рукописных книг. Для этого в большом зале собралось огромное количество писцов. Настоятель монастыря называл первое слово книги, и все писцы записывали его. Затем настоятель называл второе слово, и все писцы записывали его. Этот процесс повторялся до тех пор, пока не была прочитана вслух и переписана вся книга. На какую из систем параллельной обработки информации (см. раздел «Параллелизм на уровне процессоров») эта система больше всего похожа?
8. При продвижении сверху вниз по пятиуровневой иерархической структуре памяти время доступа возрастает. Каково отношение к времени доступа оптического диска и к регистровой памяти? (Предполагается, что диск уже вставлен.)
9. Сосчитайте скорость передачи данных в человеческом глазу, используя следующую информацию. Поле зрения состоит приблизительно из 10^6 элементов (пикселов). Каждый пиксел может сводиться к наложению трех основных цветов, каждый из которых имеет 64 степени интенсивности. Временное разрешение 100 миллисекунд.
10. Генетическая информация у всех живых существ кодируется в молекулах ДНК. Молекула ДНК представляет собой линейную последовательность четырех основных нуклеотидов: A, C, G и T. Геном человека содержит приблизительно 3×10^9 нуклеотидов в форме 100 000 генов. Какова общая информационная емкость человеческого генома (в битах)? Какова средняя информационная емкость гена (в битах)?
11. Какие из перечисленных ниже видов памяти возможны? Какие из них приемлемы? Объясните, почему.
    1) 10-битный адрес, 1024 ячейки, размер ячейки 8 битов;
    2) 10-битный адрес, 1024 ячейки, размер ячейки 12 битов;
    3) 9-битный адрес, 1024 ячейки, размер ячейки 10 битов;
    4) 11-битный адрес, 1024 ячейки, размер ячейки 10 битов;
    5) 10-битный адрес, 10 ячеек, размер ячейки 1024 бита;
    6) 1024-битный адрес, 10 ячеек, размер ячейки 10 битов.
12. Социологи могут получить 3 возможных ответа на вопрос «Верите ли вы в фей?»: да, нет, не знаю. Учитывая это, одна компьютерная компания решила создать машину для обработки данных социологических опросов. Этот компьютер имеет тринарную память, то есть каждый байт (или трайт?) состоит из 8 тритов, а каждый трит может принимать значение 0, 1 или 2. Сколько

нужно тритов для хранения 6-битного числа? Напишите выражение для числа тритов, необходимых для хранения n битов.

13. Компьютер может содержать 268 435 456 байтов памяти. Почему разработчики выбрали такое странное число вместо какого-нибудь хорошо запоминающегося, например 250 000 000?
14. Придумайте код Хэмминга для разрядов от 0 до 9.
15. Придумайте код для разрядов от 0 до 9 с интервалом Хэмминга 2.
16. В коде Хэмминга некоторые биты «пустые» в том смысле, что они используются для проверки и не несут никакой информации. Какой процент пустых битов содержится в посланиях, полная длина которых (данные + биты проверки) 2^n-1? Сосчитайте значение этого выражения при n от 3 до 10.
17. Ошибки при передаче данных по телефонной линии часто происходят «вспышками» (искажается сразу много последовательных битов). Поскольку код Хэмминга может исправлять только одиночные ошибки в символе, в данном случае он не подходит, так как шум может исказить n последовательных битов. Придумайте метод передачи текста в коде ASCII по телефонной линии, где шум может исказить 100 последовательных битов. Предполагается, что минимальный интервал между двумя искажениями составляет тысячи символов. *Подсказка*: подумайте о порядке передачи битов.
18. Сколько времени занимает считывание диска с 800 цилиндрами, каждый из которых содержит 5 дорожек по 32 сектора? Сначала считываются все сектора дорожки 0, начиная с сектора 0, затем все сектора дорожки 1, начиная с сектора 0, и т. д. Оборот совершается за 20 мс, поиск между соседними цилиндрами занимает 10 мс, а в случае расположения считываемых данных в разных частях диска — до 50 мс. Переход от одной дорожки цилиндра к другой происходит мгновенно.
19. Диск, изображенный на рис. 2.16, имеет 64 сектора на дорожке и скорость вращения 7200 оборотов в минуту. Какова скорость передачи данных на одной дорожке?
20. Компьютер содержит шину с временем цикла 25 нс. За 1 цикл он может считывать из памяти или записывать в память 32-битное слово. Компьютер имеет диск Ultra-SCSI, который использует шину и передает информацию со скоростью 40 Мбайт/с. Центральный процессор обычно вызывает из памяти и выполняет одну 32-битную команду каждые 25 нс. Насколько диск замедляет работу процессора?
21. Представьте, что вы записываете часть операционной системы, отвечающую за управление диском. Логически вы представляете себе диск как последовательность блоков от 0 на внутренней стороне до какого-либо максимума снаружи. Когда создаются файлы, вам приходится размещать свободные сектора. Вы можете двигаться от наружного края внутрь или наоборот. Имеет ли значение, какую стратегию выбрать? Поясните свой ответ.
22. Система адресации LBA использует 24 бита для обращения к сектору. Каков максимальный объем диска, с которым она может работать?

23. RAID третьего уровня может исправлять единичные битовые ошибки, используя только 1 диск четности. А что происходит в RAID-массиве второго уровня? Он ведь тоже может исправлять единичные ошибки, но использует при этом несколько дисков.
24. Какова точная емкость (в байтах) компакт-диска второго типа, содержащего данные на 74 минуты?
25. Чтобы прожигать отверстия в диске CD-R, лазер должен включаться и выключаться очень быстро. Какова длительность одного состояния (включения или выключения) в наносекундах, если компакт-диск первого типа прокручивается со скоростью 4x?
26. Чтобы вместить фильм длительностью 133 минуты на односторонний DVD с одним слоем, требуется небольшая компрессия. Вычислите, насколько нужно сжать фильм. Предполагается, что для записи дорожки изображения нужно 3,5 Гбайт, разрешающая способность изображения 720×480 пикселов с 24-битным цветом и в секунду меняется 30 кадров.
27. Скорость передачи данных между центральным процессором и связанной с ним памятью на несколько порядков выше, чем скорость передачи данных с механических устройств ввода-вывода. Каким образом это несоответствие может вызвать снижение производительности? Как можно смягчить такое снижение производительности?
28. Графический терминал имеет монитор 1024×768. Изображение на мониторе меняется 75 раз в секунду. Как часто меняется отдельный пиксел?
29. Производитель говорит, что его цветной графический терминал может воспроизводить 2^{24} различных цветов. Однако аппаратное обеспечение имеет только 1 байт для каждого пиксела. Каким же образом получается столько цветов?
30. Монохромный лазерный принтер может печатать на одном листе 50 строк по 80 символов в определенном шрифте. Символ в среднем занимает пространство 2×2 мм, причем тонер занимает 25% этого пространства, а оставшаяся часть остается белой. Толщина слоя тонера составляет 25 микрон. Картридж с тонером имеет размер 25×8×2 см. На сколько страниц хватит картриджа?
31. Когда текст в ASCII-коде с проверкой на четность передается асинхронно со скоростью 2880 символов/с через модем, передающий информацию со скоростью 28 800 бит/с, сколько процентов битов от всех полученных содержат данные?
32. Компания, выпускающая модемы, разработала новый модем с частотной модуляцией, который использует 16 частот вместо 2. Каждая секунда делится на n равных временных отрезков, каждый из которых содержит один из 16 возможных тонов. Сколько битов в секунду может передавать этот модем при использовании синхронной передачи?
33. Оцените, сколько символов (включая пробелы) содержит обычная книга по информатике. Сколько битов нужно для того, чтобы закодировать книгу

в ASCII с проверкой на четность? Сколько компакт-дисков нужно для хранения 10 000 книг по информатике? Сколько двухсторонних, двухслойных DVD-дисков нужно для хранения такого же количества книг?

34. Декодируйте следующий двоичный текст ASCII: 1001001 0100000 1001100 1001111 1010110 1000101 0100000 1011001 1001111 1010101 0101110.
35. Напишите процедуру *hamming* (*ascii*, *encoded*), которая переделывает 7 последовательных битов *ascii* в 11-битное целое кодированное число *encoded*.
36. Напишите функцию *distance* (*code*, *n*, *k*), которая на входе получает массив *code* из *n* символов по *k* битов каждый и возвращает дистанцию символа.

Глава 3
Цифровой логический уровень

В самом низу иерархической схемы на рис. 1.2 находится цифровой логический уровень, или аппаратное обеспечение компьютера. В этой главе мы рассмотрим различные аспекты цифровой логики, что должно послужить основой для изучения более высоких уровней в последующих главах. Предмет изучения находится на границе информатики и электротехники, но материал является самодостаточным, поэтому предварительного ознакомления с аппаратным обеспечением и электротехникой не потребуется.

Основные элементы, из которых конструируются цифровые компьютеры, чрезвычайно просты. Сначала мы рассмотрим эти основные элементы, а также специальную двузначную алгебру (булеву алгебру), которая используется при конструировании этих элементов. Затем мы рассмотрим основные схемы, которые можно построить из вентилей в различных комбинациях, в том числе схемы для выполнения арифметических действий. Следующая тема — как можно комбинировать вентили для хранения информации, то есть как устроена память. После этого мы перейдем к процессорам и к тому, как процессоры на одной микросхеме обмениваются информацией с памятью и периферическими устройствами. Затем мы рассмотрим различные примеры промышленного производства.

Вентили и булева алгебра

Цифровые схемы могут конструироваться из небольшого числа простых элементов путем сочетания этих элементов в различных комбинациях. В следующих разделах мы опишем эти основные элементы, покажем, как их можно сочетать, а также введем математический метод, который можно использовать при анализе их работы.

Вентили

Цифровая схема — это схема, в которой есть только два логических значения. Обычно сигнал от 0 до 1 В представляет одно значение (например, 0), а сигнал от 2 до 5 В — другое значение (например, 1). Напряжение за пределами указанных величин недопустимо. Крошечные электронные устройства, которые называются **вен-**

тилями, могут вычислять различные функции от этих двузначных сигналов. Эти вентили формируют основу аппаратного обеспечения, на которой строятся все цифровые компьютеры.

Описание принципов работы вентилей не входит в задачи этой книги, поскольку это относится к **уровню физических устройств**, который находится ниже уровня 0. Тем не менее мы очень кратко рассмотрим основной принцип, который не так уж и сложен. Вся современная цифровая логика основывается на том, что транзистор может работать как очень быстрый бинарный переключатель. На рис. 3.1, *а* изображен биполярный транзистор, встроенный в простую схему. Транзистор имеет три соединения с внешним миром: **коллектор**, **базу** и **эмиттер**. Если входное напряжение V_{in} ниже определенного критического значения, транзистор выключается и действует как очень большое сопротивление. Это приводит к выходному сигналу V_{out}, близкому к V_{cc} (напряжению, подаваемому извне), обычно +5 В для данного типа транзистора. Если V_{in} превышает критическое значение, транзистор включается и действует как провод, вызывая заземление сигнала V_{out} (по соглашению 0 В).

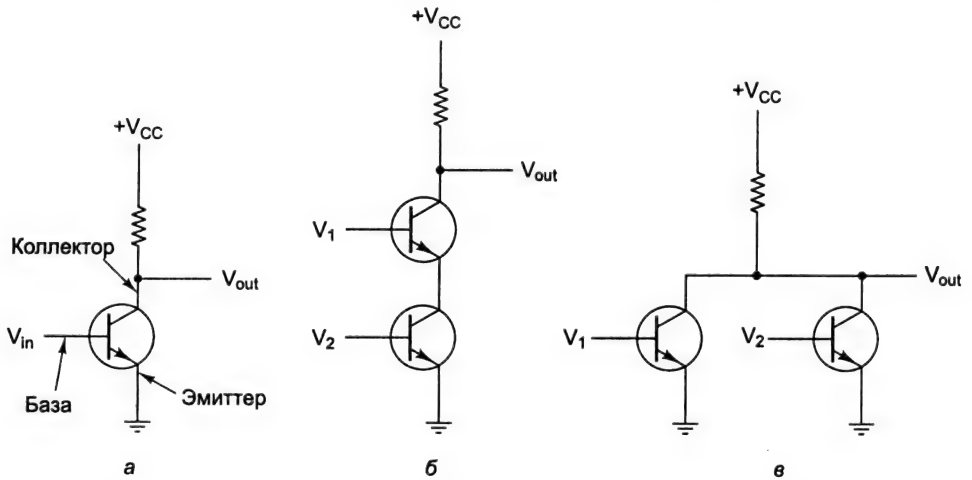


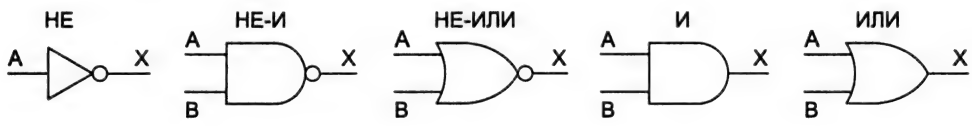
Рис. 3.1. Транзисторный инвертор (*а*); вентиль НЕ-И (*б*); вентиль НЕ-ИЛИ (*в*)

Важно отметить, что если напряжение V_{in} низкое, то V_{out} высокое, и наоборот. Эта схема, таким образом, является инвертором, превращающим логический 0 в логическую 1 и логическую 1 в логический 0. Резистор (ломаная линия) нужен для ограничения количество тока, проходящего через транзистор, чтобы транзистор не сгорел. На переключение с одного состояния на другое обычно требуется несколько наносекунд.

На рис. 3.1, *б* два транзистора соединены последовательно. Если и напряжение V_1, и напряжение V_2 высокое, то оба транзистора будут служить проводниками и снижать V_{out}. Если одно из входных напряжений низкое, то соответствующий транзистор будет выключаться и напряжение на выходе будет высоким. Другими словами, V_{out} будет низким тогда и только тогда, когда и напряжение V_1, и напряжение V_2 высокое.

На рис. 3.1, *в* два транзистора соединены параллельно. Если один из входных сигналов высокий, будет включаться соответствующий транзистор и снижать выходной сигнал. Если оба напряжения на входе низкие, то выходное напряжение будет высоким.

Эти три схемы образуют три простейших вентиля. Они называются вентилями НЕ, НЕ-И и НЕ-ИЛИ. Вентили НЕ часто называют **инверторами**. Мы будем использовать оба термина. Если мы примем соглашение, что высокое напряжение (V_{cc}) — это логическая 1, а низкое напряжение («земля») — логический 0, то мы сможем выражать значение на выходе как функцию от входных значений. Значки, которые используются для изображения этих трех типов вентилей, показаны на рис. 3.2, *а* — *в*. Там же приводится поведение функции для каждой схемы. На этих рисунках A и B — это входные сигналы, а X — выходной сигнал. Каждая строка таблицы определяет выходной сигнал для различных комбинаций входных сигналов.



A	X
0	1
1	0

а

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

б

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

в

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

г

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

д

Рис. 3.2. Значки для изображения 5 основных вентилей. Поведение функции для каждого вентиля

Если выходной сигнал (см. рис. 3.1, *б*) подать в инвертор, мы получим другую схему, противоположную вентилю НЕ-И, то есть такую схему, у которой выходной сигнал равен 1 тогда и только тогда, когда оба входных сигнала равны 1. Такая схема называется вентилем И; ее схематическое изображение и описание соответствующей функции даны на рис. 3.2, *г*. Точно так же вентиль НЕ-ИЛИ может быть связан с инвертором. Тогда получится схема, у которой выходной сигнал равен 1 в том случае, если хотя бы один из входных сигналов — 1, и равен 0, если оба входных сигнала равны 0. Изображение этой схемы, которая называется вентилем ИЛИ, а также описание соответствующей функции даны на рис. 3.2, *д*. Маленькие кружочки в схемах инвертора, вентиля НЕ-И и вентиля НЕ-ИЛИ называются **инвертирующими выходами**. Они также могут использоваться в другом контексте для указания на инвертированный сигнал.

Пять вентилей, изображенных на рис. 3.2, составляют основу цифрового логического уровня. Из предшествующего обсуждения должно быть ясно, что вентили НЕ-И и НЕ-ИЛИ требуют два транзистора каждый, а вентили И и ИЛИ — три транзистора каждый. По этой причине во многих компьютерах используются вен-

тили НЕ-И и НЕ-ИЛИ, а не И и ИЛИ. (На практике все вентили выполняются несколько по-другому, но НЕ-И и НЕ-ИЛИ все равно проще, чем И и ИЛИ.) Следует упомянуть, что вентили могут иметь более двух входов. В принципе вентиль НЕ-И, например, может иметь произвольное количество входов, но на практике больше восьми обычно не бывает.

Хотя устройство вентилей относится к уровню физических устройств, мы все же упомянем основные серии производственных технологий, так как они часто упоминаются в литературе. Две основные технологии — **биполярная** и **МОП** (металл-оксид-полупроводник). Среди биполярных технологий можно назвать **ТТЛ** (транзисторно-транзисторную логику), которая служила основой цифровой электроники на протяжении многих лет, и **ЭСЛ** (эмиттерно-связанную логику), которая используется в тех случаях, когда требуется высокая скорость выполнения операций.

Вентили МОП работают медленнее, чем ТТЛ и ЭСЛ, но потребляют гораздо меньше энергии и занимают гораздо меньше места, поэтому можно компактно расположить большое количество таких вентилей. Вентили МОП имеют несколько разновидностей: p-канальный МОП-прибор, n-канальный МОП-прибор и комплиментарный МОП. Хотя МОП-транзисторы конструируются не так, как биполярные транзисторы, они обладают такой же способностью функционировать, как электронные переключатели. Современные процессоры и память чаще всего производятся с использованием технологии комплиментарных МОП, которая работает при напряжении +3,3 В. Это все, что мы можем сказать об уровне физических устройств. Читатели, желающие узнать больше об этом уровне, могут обратиться к литературе, приведенной в главе 9.

Булева алгебра

Чтобы описать схемы, которые строятся путем сочетания различных вентилей, нужен особый тип алгебры, в которой все переменные и функции могут принимать только два значения: 0 и 1. Такая алгебра называется **булевой алгеброй**. Она названа в честь английского математика Джорджа Буля (1815–1864). На самом деле в данном случае мы говорим об особом типе булевой алгебры, а именно об **алгебре релейных схем**, но термин «булева алгебра» очень часто используется в значении «алгебра релейных схем», поэтому мы не будем их различать.

Как и в обычной алгебре (то есть в той, которую изучают в школе), в булевой алгебре есть свои функции. Булева функция имеет одну или несколько переменных и выдает результат, который зависит только от значений этих переменных. Можно определить простую функцию f, сказав, что f(A)=1, если A=0, и f(A)=0, если A=1. Такая функция будет функцией НЕ (см. рис. 3.2, *а*).

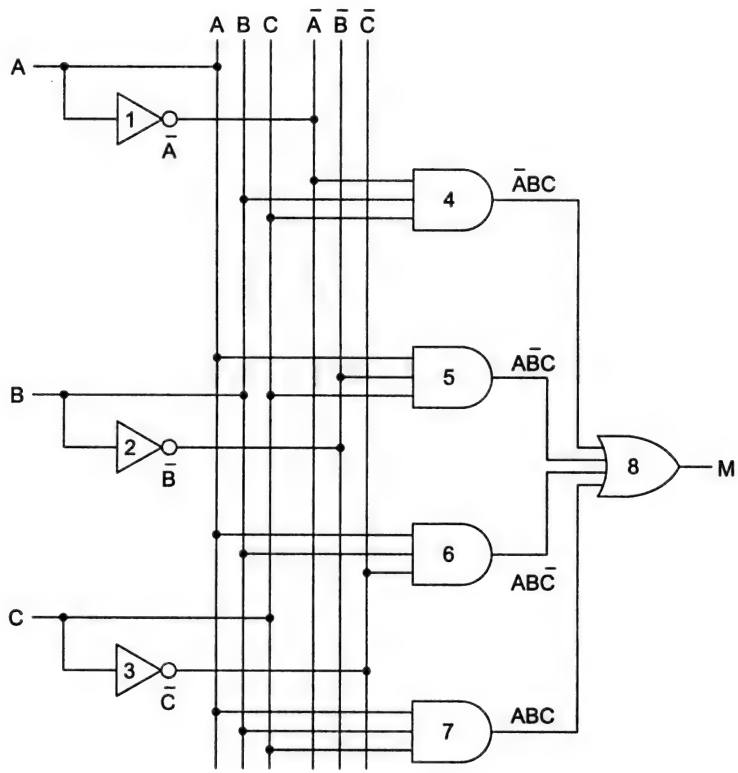
Так как булева функция от n переменных имеет только 2^n возможных комбинаций значений переменных, то такую функцию можно полностью описать в таблице с 2^n строками. В каждой строке будет даваться значение функции для разных комбинаций значений переменных. Такая таблица называется **таблицей истинности**. Все таблицы на рис. 3.2 представляют собой таблицы истинности. Если мы

договоримся всегда располагать строки таблицы истинности по порядку номеров, то есть для двух переменных в порядке 00, 01, 10, 11, то функцию можно полностью описать 2^n-битным двоичным числом, которое получается, если считывать по вертикали колонку результатов в таблице истинности. Таким образом, НЕ-И — это 1110, НЕ-ИЛИ — 1000, И — 0001 и ИЛИ — 0111. Очевидно, что существует только 16 булевых функций от двух переменных, которым соответствуют 16 возможных 4-битных цепочек. В обычной алгебре, напротив, есть бесконечное число функций от двух переменных, и ни одну из них нельзя описать, дав таблицу значений этой функции для всех возможных значений переменных, поскольку каждая переменная может принимать бесконечное число значений.

На рис. 3.3, *а* показана таблица истинности для булевой функции от трех переменных: M=f(A, B, C). Это функция большинства, которая принимает значение 0, если большинство переменных равно 0, и 1, если большинство переменных равно 1. Хотя любая булева функция может быть определена с помощью таблицы истинности, с возрастанием количества переменных такой тип записи становится громоздким. Поэтому вместо таблиц истинности часто используется другой тип записи.

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

а



б

Рис. 3.3. Таблица истинности для функции большинства от трех переменных (*а*); схема для этой функции (*б*)

Чтобы увидеть, каким образом осуществляется этот другой тип записи, отметим, что любую булеву функцию можно определить, указав, какие комбинации значений переменных дают значение функции 1. Для функции, приведенной на рис. 3.3, *а*, существует 4 комбинации переменных, которые дают значение функции 1. Мы будем рисовать черту над переменной, чтобы показать, что ее значение инвертируется. Отсутствие черты означает, что значение переменной не инвертируется. Кроме того, мы будем использовать знак умножения (точку) для обозначения булевой функции И (знак умножения может опускаться) и + для обозначения булевой функции ИЛИ. Например, $A\overline{B}C$ принимает значение 1, только если A=1, B=0 и C=1. $A\overline{B}+B\overline{C}$ принимает значение 1, только если (A=1 и B=0) или (B=1 и C=0). В таблице на рис. 3.3, *а* функция принимает значение 1 в четырех строках: $\overline{A}BC$, $A\overline{B}C$, $AB\overline{C}$ и ABC. Функция M принимает значение истины (то есть 1), если одно из этих четырех условий истинно. Следовательно, мы можем написать

$M=\overline{A}BC+A\overline{B}C+AB\overline{C}+ABC$.

Это компактная запись таблицы истинности. Таким образом, функцию от n переменных можно описать суммой максимум 2^n произведений, при этом в каждом произведении будет по n множителей. Как мы скоро увидим, такая формулировка особенно важна, поскольку она ведет прямо к реализации данной функции с использованием стандартных вентилей.

Важно понимать различие между абстрактной булевой функцией и ее реализацией с помощью электронной схемы. Булева функция состоит из переменных, например A, B и C, и операторов И, ИЛИ и НЕ. Булева функция описывается с помощью таблицы истинности или специальной записи, например:

$F=A\overline{B}C+AB\overline{C}$.

Булева функция может реализовываться с помощью электронной схемы (часто различными способами) с использованием сигналов, которые представляют входные и выходные переменные, и вентилей, например, И, ИЛИ и НЕ.

Реализация булевых функций

Как было сказано выше, представление булевой функции в виде суммы максимум 2^n произведений делает возможной реализацию этой функции. На рисунке 3.3 можно увидеть, как это осуществляется. На рисунке 3.3, *б* входные сигналы A, B и C показаны с левой стороны, а функция M, полученная на выходе, показана с правой стороны. Поскольку необходимы дополнительные величины (инверсии) входных переменных, они образуются путем провода сигнала через инверторы 1, 2 и 3. Чтобы сделать рисунок понятней, мы нарисовали 6 вертикальных линий, 3 из которых связаны с входными переменными, а 3 другие — с их инверсиями. Эти линии обеспечивают передачу входного сигнала к вентилям. Например, вентили 5, 6 и 7 в качестве входа используют A. В реальной схеме эти вентили, вероятно, будут непосредственно соединены проводом с A без каких-либо промежуточных вертикальных проводов.

Схема содержит четыре вентиля И, по одному для каждого члена в уравнении для M (то есть по одному для каждой строки в таблице истинности с результатом 1). Каждый вентиль И вычисляет одну из указанных строк таблицы истинно-

ти. В конце концов все данные произведения суммируются (имеется в виду операция ИЛИ) для получения конечного результата.

Посмотрите на рис. 3.3, *б*. В этой книге мы будем использовать следующее соглашение: если две линии на рисунке пересекаются, связь подразумевается только в том случае, если на пересечении указана жирная точка. Например, выход вентиля 3 пересекает все 6 вертикальных линий, но связан он только с $\overline{C}$. Отметим, что другие авторы могут использовать другие соглашения.

Из рисунка 3.3 должно быть ясно, как реализовать схему для любой булевой функции:

1. Составить таблицу истинности для данной функции.
2. Обеспечить инверторы, чтобы порождать инверсии для каждого входного сигнала.
3. Нарисовать вентиль И для каждой строки таблицы истинности с результатом 1.
4. Соединить вентили И с соответствующими входными сигналами.
5. Вывести выходы всех вентилей И в вентиль ИЛИ.

Мы показали, как реализовать любую булеву функцию с использованием вентилей НЕ, И и ИЛИ. Однако гораздо удобнее строить схемы с использованием одного типа вентилей. К счастью, можно легко преобразовать схемы, построенные по предыдущему алгоритму, в форму НЕ-И или НЕ-ИЛИ. Чтобы осуществить такое преобразование, все, что нам нужно, — это способ воплощения НЕ, И и ИЛИ с помощью одного типа вентилей. На рисунке 3.4 показано, как это можно сделать, используя только вентили НЕ-И или только вентили НЕ-ИЛИ. Отметим, что существуют также другие способы подобного преобразования.

Для того чтобы реализовать булеву функцию с использованием только вентилей НЕ-И или только вентилей НЕ-ИЛИ, можно сначала следовать алгоритму, описанному выше, и сконструировать схему с вентилями НЕ и И и ИЛИ. Затем нужно заменить многовходовые вентили эквивалентными схемами с использованием двухвходовых вентилей. Например, A+B+C+D можно поменять на (A+B)+(C+D), используя три двухвходовых вентиля. Затем вентили НЕ и И и ИЛИ заменяются схемами, изображенными на рис. 3.4.

Хотя такая процедура и не приводит к оптимальным схемам с точки зрения минимального числа вентилей, она демонстрирует, что подобное преобразование осуществимо. Вентили НЕ-И и НЕ-ИЛИ считаются **полными**, потому что можно вычислить любую булеву функцию, используя только вентили НЕ-И или только вентили НЕ-ИЛИ. Ни один другой вентиль не обладает таким свойством, вот почему именно эти два типа вентилей предпочтительны при построении схем.

Эквивалентность схем

Разработчики схем часто стараются сократить число вентилей, чтобы снизить цену, уменьшить занимаемое схемой место, сократить потребление энергии и т. д. Чтобы упростить схему, разработчик должен найти другую схему, которая может вычислять ту же функцию, но при этом требует меньшего количества вентилей (или может работать с более простыми вентилями, например двухвходовыми вместо четырехвходовых). Булева алгебра является ценным инструментом в поиске эквивалентных схем.

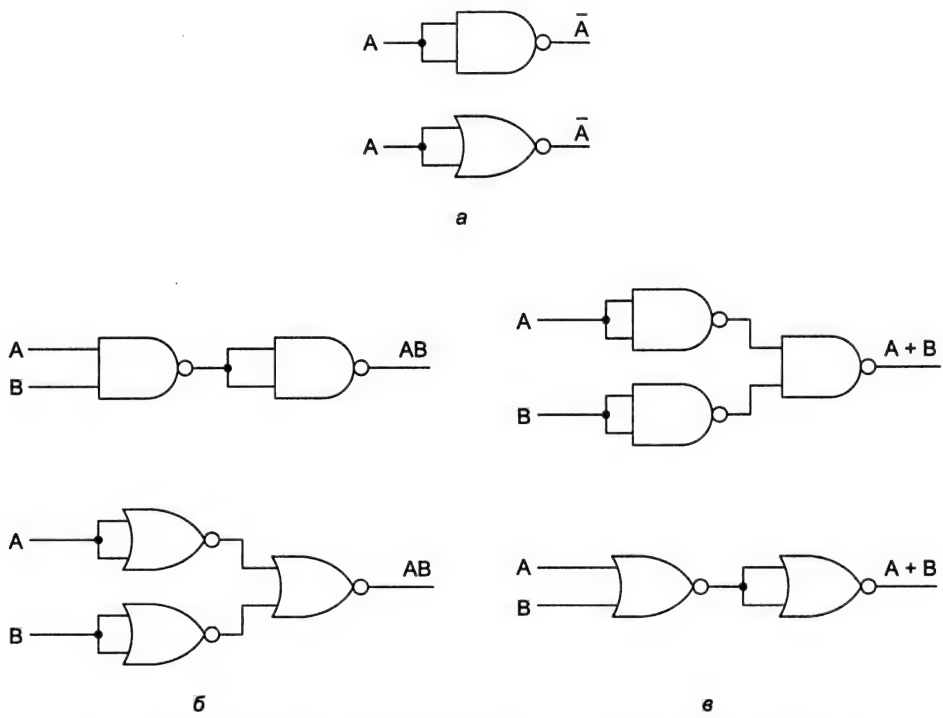
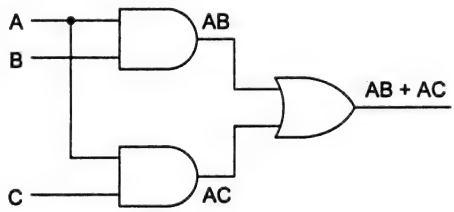


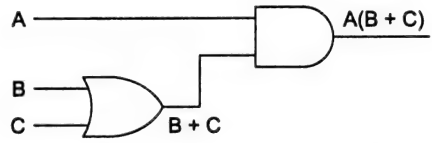
Рис. 3.4. Конструирование вентилей НЕ (*а*), И (*б*) и ИЛИ (*в*) с использованием только вентилей НЕ-И или только вентилей НЕ-ИЛИ

В качестве примера использования булевой алгебры рассмотрим схему и таблицу истинности для AB+AC (рис. 3.5, *а*). Хотя мы это еще не обсуждали, многие правила обычной алгебры имеют силу для булевой алгебры. Например, выражение AB+AC может быть преобразовано в A(B+C) с помощью дистрибутивного закона. На рис. 3.5, *б* показана схема и таблица истинности для A(B+C). Две функции являются эквивалентными тогда и только тогда, когда обе функции принимают одно и то же значение для всех возможных переменных. Из таблиц истинности на рис. 3.5 ясно видно, что A(B+C) эквивалентно AB+AC. Несмотря на эту эквивалентность, схема на рис. 3.5, *б* лучше, чем схема на рис. 3.5, *а*, поскольку она содержит меньше вентилей.

Обычно разработчик исходит из определенной булевой функции, а затем применяет к ней законы булевой алгебры, чтобы найти более простую функцию, эквивалентную исходной. На основе полученной функции можно конструировать схему.

Чтобы использовать данный подход, нам нужны некоторые равенства из булевой алгебры. В табл. 3.1 показаны некоторые основные законы. Интересно отметить, что каждый закон имеет две формы. Одну форму из другой можно получить, меняя И на ИЛИ и 0 на 1. Все законы можно легко доказать, составив их таблицы истинности. Почти во всех случаях результаты очевидны, за исключением законов Де Моргана, законов поглощения и дистрибутивного закона A+BC=(A+B)(A+C). Законы Де Моргана распространяются на выражения с более чем двумя переменными, например $\overline{ABC}=\overline{A}+\overline{B}+\overline{C}$.





A	B	C	AB	AC	AB + AC
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

а

A	B	C	A	B + C	A(B + C)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

б

Рис. 3.5. Две эквивалентные функции: AB+AC (*а*); A(B+C) (*б*).

Таблица 3.1. Некоторые законы булевой алгебры

Названия законов	И	ИЛИ
Законы тождества	1A=A	0+A=A
Законы нуля	0A=0	1+A=1
Законы идемпотентности	AA=A	A+A=A
Законы инверсии	$A\overline{A}=0$	$A+\overline{A}=1$
Коммуникативные законы	AB=BA	A+B=B+A
Ассоциативные законы	(AB)C=A(BC)	(A+B)+C=A+(B+C)
Дистрибутивные законы	A+BC=(A+B)(A+C)	A(B+C)=AB+AC
Законы поглощения	A(A+B)=A	A+AB=A
Законы Де Моргана	$\overline{AB}=\overline{A}+\overline{B}$	$\overline{A+B}=\overline{A}\,\overline{B}$

Законы Де Моргана предполагают альтернативную запись. На рис. 3.6, *а* форма И дается с отрицанием, которое показывается с помощью инвертирующих входов и выходов. Таким образом, вентиль ИЛИ с инвертированными входными сигналами эквивалентен вентилю НЕ-И. Из рис. 3.6, *б*, на котором изображена вторая форма закона Де Моргана, ясно, что вместо вентиля НЕ-ИЛИ можно нарисовать вентиль И с инвертированными входами. С помощью отрицания обеих форм закона Де Моргана мы приходим к эквивалентным репрезентациям вентилей И и ИЛИ (см. рис. 3.6, *в* и 3.6, *г*). Аналогичные символические изображения существуют для различных форм закона Де Моргана (например, n-входовый вентиль НЕ-И становится вентилем ИЛИ с инвертированными входами).

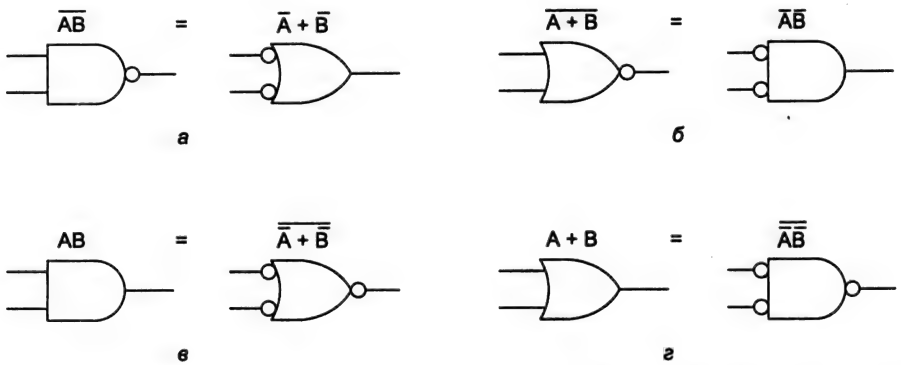
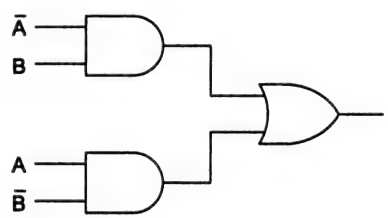


Рис. 3.6. Альтернативные обозначения некоторых вентилей: НЕ-И (*а*); НЕ-ИЛИ (*б*); И (*в*); ИЛИ (*г*)

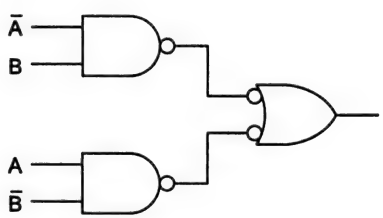
Используя уравнения, указанные на рис. 3.6, и аналогичные уравнения для многовходовых вентилей, можно легко преобразовать сумму произведений в чистую форму НЕ-И или чистую форму НЕ-ИЛИ. В качестве примера рассмотрим функцию ИСКЛЮЧАЮЩЕЕ ИЛИ (рис. 3.7, *а*). Стандартная схема, выражающая сумму произведений, показана на рис. 3.7, *б*. Чтобы перейти к форме НЕ-И, нужно линии, соединяющие выходы вентилей И с входом вентиля ИЛИ, нарисовать с инвертирующими входами и выходами, как показано на рис. 3.7, *в*. Затем, применяя рис. 3.6, *а*, мы приходим к рис. 3.7, *г*. Переменные $\overline{A}$ и $\overline{B}$ можно получить из A и B, используя вентили НЕ-И или НЕ-ИЛИ с объединенными входами. Отметим, что инвертирующие входы (выходы) могут перемещаться вдоль линии по желанию, например, от выходов входных вентилей к входам выходного вентиля.

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

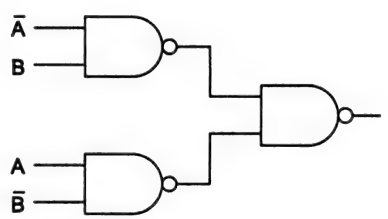
а



б



в



г

Рис. 3.7. Таблица истинности для функции ИСКЛЮЧАЮЩЕЕ ИЛИ (*а*); Три схемы для вычисления этой функции (*б*), (*в*), (*г*)

Очень важно отметить, что один и тот же вентиль может вычислять разные функции в зависимости от используемых соглашений. На рис. 3.8, *а* мы показали выход определенного вентиля, F, для различных комбинаций входных сигналов. И входные, и выходные сигналы показаны в вольтах. Если мы примем соглашение, что 0 В — это логический ноль, а 3,3 В или 5 В — логическая единица, мы получим таблицу истинности, показанную на рис. 3.8, *б*, то есть функцию И. Такое соглашение называется **позитивной логикой**. Однако если мы примем **негативную логику**, то есть условимся, что 0 В — это логическая единица, а 3,3 В или 5 В — логический ноль, то мы получим таблицу истинности, показанную на рис. 3.8, *в*, то есть функцию ИЛИ.

A	B	F
0^V	0^V	0^V
0^V	5^V	0^V
5^V	0^V	0^V
5^V	5^V	5^V

а

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

б

A	B	F
1	1	1
1	0	1
0	1	1
0	0	0

в

Рис. 3.8. Электрические характеристики устройства (*а*); позитивная логика (*б*); негативная логика (*в*)

Таким образом, все зависит от того, какое соглашение выбрано для отображения вольт в логических величинах. В этой книге мы будем использовать позитивную логику. Случаи использования негативной логики будут оговариваться отдельно.

Основные цифровые логические схемы

В предыдущих разделах мы увидели, как реализовать простейшие схемы с использованием отдельных вентилей. На практике в настоящее время схемы очень редко конструируются вентиль за вентилем, хотя когда-то это было распространено. Сейчас стандартные блоки представляют собой модули, которые содержат ряд вентилей. В следующих разделах мы рассмотрим эти стандартные блоки более подробно и увидим, как они используются и как их можно построить из отдельных вентилей.

Интегральные схемы

Вентили производятся и продаются не по отдельности, а в модулях, которые называются **интегральными схемами (ИС)** или **микросхемами**. Интегральная схема представляет собой квадратный кусочек кремния размером примерно 5×5 мм, на котором находится несколько вентилей[1]. Маленькие интегральные схемы обычно

[1] Следует заметить, что эти сведения относятся к семидесятым годам прошлого века. В настоящее время степень интеграции стала выше на несколько порядков, и такие простейшие интегральные схемы в вычислительной технике уже давно не используются. — *Примеч. научн. ред.*

помещаются в прямоугольные пластиковые или керамические корпуса размером от 5 до 15 мм в ширину и от 20 до 50 мм в длину. Вдоль длинных сторон располагается два параллельных ряда выводов около 5 мм в длину, которые можно втыкать в разъемы или впаивать в печатную плату. Каждый вывод соединяется с входом или выходом какого-нибудь вентиля, или с источником питания, или с «землей». Корпус с двумя рядами выводов снаружи и интегральными схемами внутри официально называется **двурядным корпусом (Dual Inline Package**, сокращенно **DIP**), но все называют его микросхемой, стирая различие между куском кремния и корпусом, в который он помещается. Большинство корпусов имеют 14, 16, 18, 20, 22, 24, 28, 40, 64 или 68 выводов. Для больших микросхем часто используются корпуса, у которых выводы расположены со всех четырех сторон или снизу.

Микросхемы можно разделить на несколько классов с точки зрения количества вентилей, которые они содержат. Эта классификация, конечно, очень грубая, но иногда она может быть полезна:

- ✦ МИС (малая интегральная схема): от 1 до 10 вентилей.
- ✦ СИС (средняя интегральная схема): от 1 до 100 вентилей.
- ✦ БИС (большая интегральная схема): от 100 до 100 000 вентилей.
- ✦ СБИС (сверхбольшая интегральная схема): более 100 000 вентилей.

Эти схемы имеют различные свойства и используются для различных целей.

МИС обычно содержит от двух до шести независимых вентилей, каждый из которых может использоваться отдельно, как описано в предыдущих разделах. На рис. 3.9 изображена обычная микросхема МИС, содержащая четыре вентиля НЕ-И. Каждый из этих вентилей имеет два входа и один выход, что требует наличия 12 выводов. Кроме того, микросхеме требуется питание (V_{cc}) и «земля» (GND). Они разделяются всеми вентилями. На корпусе рядом с выводом 1 обычно имеется паз, чтобы можно было определить, что это вывод 1. Чтобы избежать путаницы на диаграмме, по соглашению не показываются неиспользованные вентили, источник питания и «земля».

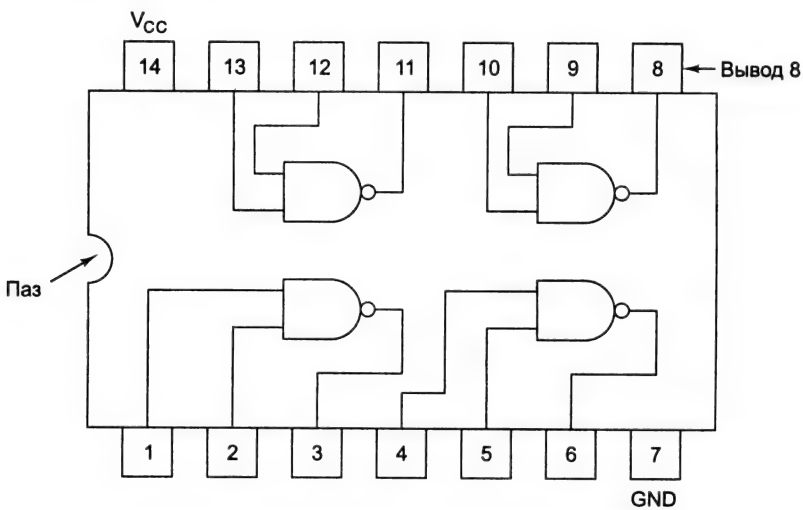


Рис. 3.9. Микросхема МИС, содержащая 4 вентиля

Подобные микросхемы стоят несколько центов. Каждая микросхема МИС содержит несколько вентилей и примерно до 20 выводов. В 70-е годы компьютеры конструировались из большого числа таких микросхем, но в настоящее время на одну микросхему помещается целый центральный процессор и существенная часть памяти (кэш-памяти).

Для удобства мы считаем, что у вентиля появляются изменения на выходе, как только появляются изменения на входе. На самом деле существует определенная **задержка вентиля**, которая включает в себя время прохождения сигнала через микросхему и время переключения. Время задержки обычно составляет от 1 до 10 нс.

В настоящее время стало возможным помещать до 10 млн транзисторов на одну микросхему[1]. Так как любая схема может быть сконструирована из вентилей НЕ-И, может создаться впечатление, что производитель способен изготовить микросхему, содержащую 5 млн вентилей НЕ-И. К несчастью, для создания такой микросхемы потребуется 15 000 002 выводов. Поскольку стандартный вывод занимает 0,1 дюйм, микросхема будет более 18 км в длину, что отрицательно скажется на покупательной способности. Поэтому чтобы использовать преимущество данной технологии, нужно разработать такие схемы, у которых количество вентилей сильно превышает количество выводов. В следующих разделах мы рассмотрим простые микросхемы МИС, в которых несколько вентилей соединены определенным образом между собой для вычисления некоторой функции, но при этом требуется небольшое число внешних выводов.

Комбинационные схемы

Многие применения цифровой логики требуют наличия схем с несколькими входами и несколькими выходами, в которых выходные сигналы определяются текущими входными сигналами. Такая схема называется **комбинационной схемой**. Не все схемы обладают таким свойством. Например, схема, содержащая элементы памяти, может генерировать выходные сигналы, которые зависят от значений, хранящихся в памяти. Микросхема, которая реализует таблицу истинности (например, приведенную на рис. 3.3, *а*), является типичным примером комбинационной схемы. В этом разделе мы рассмотрим наиболее часто используемые комбинационные схемы.

Мультиплексоры

На цифровом логическом уровне **мультиплексор** представляет собой схему с 2^n входами, одним выходом и n линиями управления, которые выбирают один из входов. Выбранный вход соединяется с выходом. На рис. 3.10 изображена схема восьмивходового мультиплексора. Три линии управления A, B и C кодируют 3-битное число, которое указывает, какая из восьми линий входа должна соединяться с вентилем ИЛИ и, следовательно, с выходом. Вне зависимости от того, какое значение будет на линиях управления, семь вентилей И будут всегда выдавать на выходе 0, а оставшийся может выдавать или 0, или 1 в зависимости от значения

[1] Не стоит забывать закон Мура. Ядро процессора Pentium IV содержит уже 42 млн транзисторов, и очевидно, это не предел. — *Примеч. научн. ред.*

выбранной линии входа. Каждый вентиль И запускается определенной комбинацией линий управления. Схема мультиплексора показана на рис. 3.10. Если к этому добавить источник питания и «землю», то мультиплексор можно запаковать в корпус с 14 выводами.

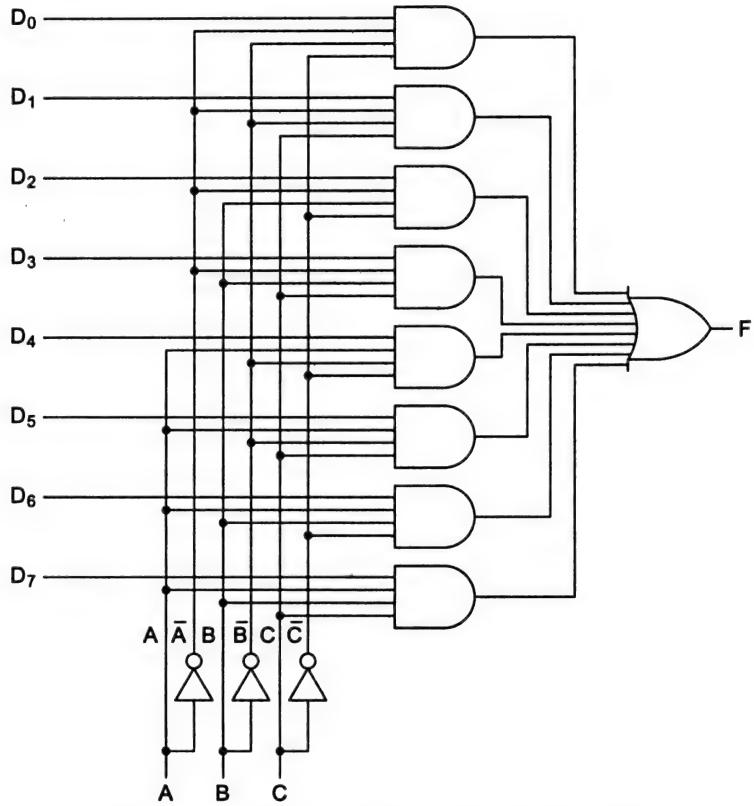


Рис. 3.10. Схема восьмивходового мультиплексора

Используя мультиплексор, мы можем реализовать функцию большинства (см. рис. 3.3, *а*), как показано на рис. 3.11, б. Для каждой комбинации A, B и C выбирается одна из входных линий. Каждый вход соединяется или с V_{cc} (логическая 1), или с «землей» (логический 0). Алгоритм соединения входов очень прост: входной сигнал D_i такой же, как значение в строке i в таблице истинности. На рис. 3.3, *а* в строках 0, 1, 2 и 4 значение функции равно 0, поэтому соответствующие входы заземляются; в оставшихся строках значение функции равно 1, поэтому соответствующие входы соединяются с логической 1. Таким способом можно реализовать любую таблицу истинности с тремя переменными, используя микросхему на рис. 3.11, *а*.

Мы уже видели, как мультиплексор может использоваться для выбора одного из нескольких входов и как он может реализовать таблицу истинности. Его также можно использовать в качестве преобразователя параллельного кода в последова-

тельный. Если подать 8 битов данных на линии входа, а затем переключать линии управления последовательно от 000 до 111 (это двоичные числа), 8 битов поступят на линию выхода последовательно. Обычно такое преобразование осуществляется при вводе информации с клавиатуры, поскольку каждое нажатие клавиши определяет 7- или 8-битное число, которое должно передаваться последовательно по телефонной линии.

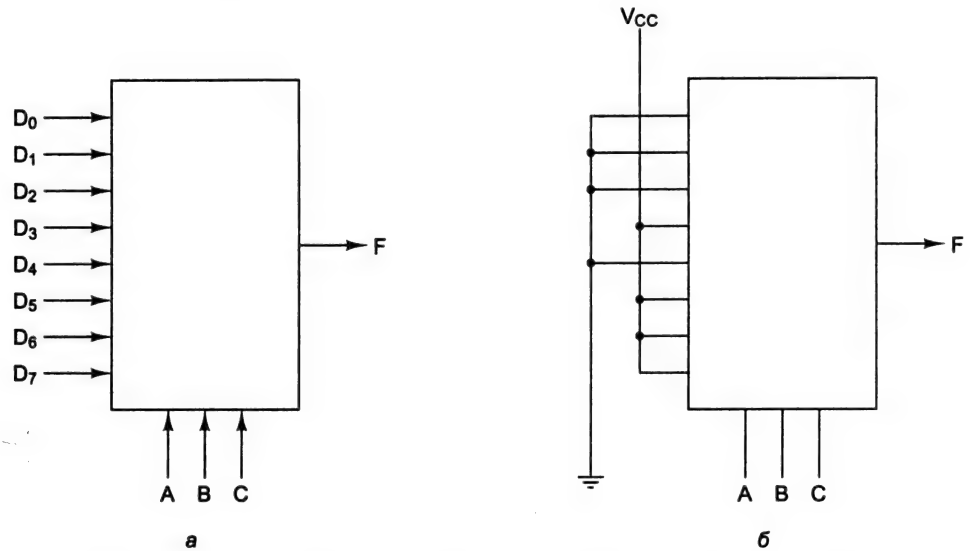


Рис. 3.11. Мультиплексор, построенный на СИС (*а*); тот же мультиплексор, смонтированный для вычисления функции большинства (*б*)

Противоположностью мультиплексора является **демультиплексор**, который соединяет единственный входной сигнал с одним из 2^n выходов в зависимости от значений n линий управления. Если бинарное значение линий управления равно k, то выбирается выход k.

Декодеры

В качестве второго примера рассмотрим схему, которая получает на входе n-битное число и использует его для того, чтобы выбрать (то есть установить на значение 1) одну из 2^n выходных линий. Такая схема называется **декодером**. Пример декодера для n=3 показан на рис. 3.12.

Чтобы понять, зачем нужен декодер, представим себе память, состоящую из 8 микросхем, каждая из которых содержит 1 Мбайт. Микросхема 0 имеет адреса от 0 до 1 Мбайт, микросхема 1 — адреса от 1 Мбайт до 2 Мбайт и т. д. Три старших двоичных разряда адреса используются для выбора одной из восьми микросхем. На рис. 3.12 эти три бита — три входа A, B и C. В зависимости от входных сигналов ровно одна из восьми выходных линий (D_0, ..., D_7) принимает значение 1; остальные линии принимают значение 0. Каждая выходная линия запускает одну из восьми микросхем памяти. Поскольку только одна линия принимает значение 1, запускается только одна микросхема.

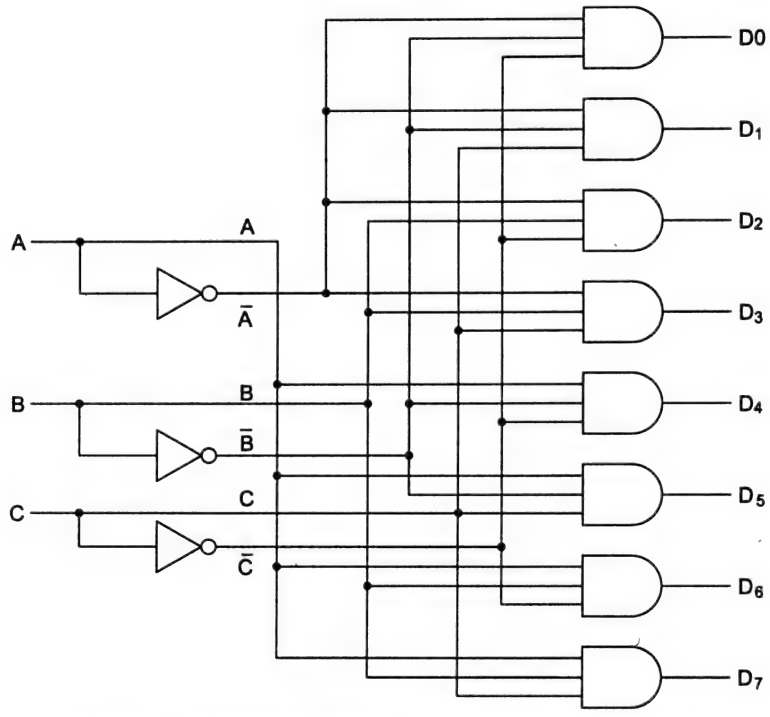


Рис. 3.12. Схема декодера, содержащего 3 входа и 8 выходов

Принцип работы схемы, изображенной на рис. 3.12, не сложен. Каждый вентиль И имеет три входа, из которых первый или A, или $\overline{A}$, второй или B, или $\overline{B}$, а третий или C, или $\overline{C}$. Каждый вентиль запускается различной комбинацией входов: D_0 — сочетанием $\overline{A}\,\overline{B}\,\overline{C}$, D_1 — $\overline{A}\,\overline{B}\,C$ и т. д.

Компараторы

Еще одна полезная схема — **компаратор**. Компаратор сравнивает два слова, которые поступают на вход. Компаратор, изображенный на рис. 3.13, принимает два входных сигнала, A и B, каждый длиной 4 бита, и выдает 1, если они равны, и 0, если они не равны. Схема основывается на вентиле ИСКЛЮЧАЮЩЕЕ ИЛИ, который выдает 0, если сигналы на входе равны, и 1, если сигналы на входе не равны. Если все четыре входных слова равны, все четыре вентиля ИСКЛЮЧАЮЩЕЕ ИЛИ должны выдавать 0. Эти четыре сигнала затем поступают в вентиль ИЛИ. Если в результате получается 0, значит, слова, поступившие на вход, равны; в противном случае они не равны. В нашем примере мы использовали вентиль ИЛИ в качестве конечной стадии, чтобы поменять значение полученного результата: 1 означает равенство, а 0 — неравенство.

Программируемые логические матрицы

Ранее мы рассказывали, что любую функцию (таблицу истинности) можно представить в виде суммы произведений и, следовательно, воплотить в схеме, исполь-

зуя вентили И и ИЛИ. Для вычисления сумм произведений служит так называемая **программируемая логическая матрица** (рис. 3.14). Эта микросхема содержит входы для 12 переменных. Дополнительные сигналы (инверсии) генерируются внутри самой микросхемы. В итоге всего получается 24 входных сигнала. Какой именно входной сигнал поступает в определенный вентиль И, определяется по матрице 24×50 бит. Каждая из входных линий к 50 вентилям И содержит плавкую перемычку. При выпуске с завода все 1200 перемычек остаются нетронутыми. Чтобы запрограммировать матрицу, покупатель выжигает выбранные перемычки, прикладывая к схеме высокое напряжение.

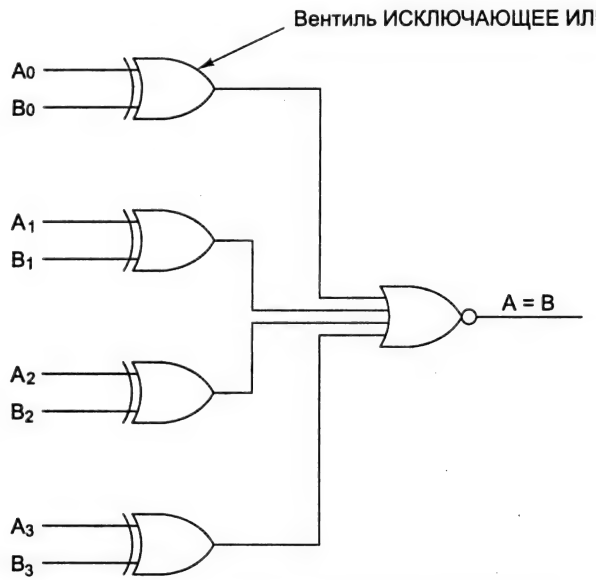


Рис. 3.13. Простой четырехразрядный компаратор

Выходная часть схемы состоит из шести вентилей ИЛИ, каждый из которых содержит до 50 входов, что соответствует наличию 50 выходов у вентилей И. Какие из потенциально возможных связей действительно существуют, зависит от того, как была запрограммирована матрица 50×6. Микросхема имеет 12 входных выводов, 6 выходных выводов, питание и «землю» (то есть всего 20 выводов).

Приведем пример использования программируемой логической матрицы. Рассмотрим схему, изображенную на рис. 3.3, *б*. Она содержит три входа, четыре вентиля И, один вентиль ИЛИ и три инвертора. Если запрограммировать нашу матрицу определенным образом, она сможет вычислять ту же функцию, используя три из 12 входов, четыре из 50 вентилей И и один из 6 вентилей ИЛИ. (Четыре вентиля И должны вычислять $\overline{A}BC$, $A\overline{B}C$, $AB\overline{C}$ и ABC; вентиль ИЛИ принимает эти 4 произведения в качестве входных данных.) Можно сделать так, чтобы та же программируемая логическая матрица вычисляла одновременно сумму четырех функций одинаковой сложности. Для простых функций ограничивающим фактором является число входных переменных, для более сложных — вентили И и ИЛИ.

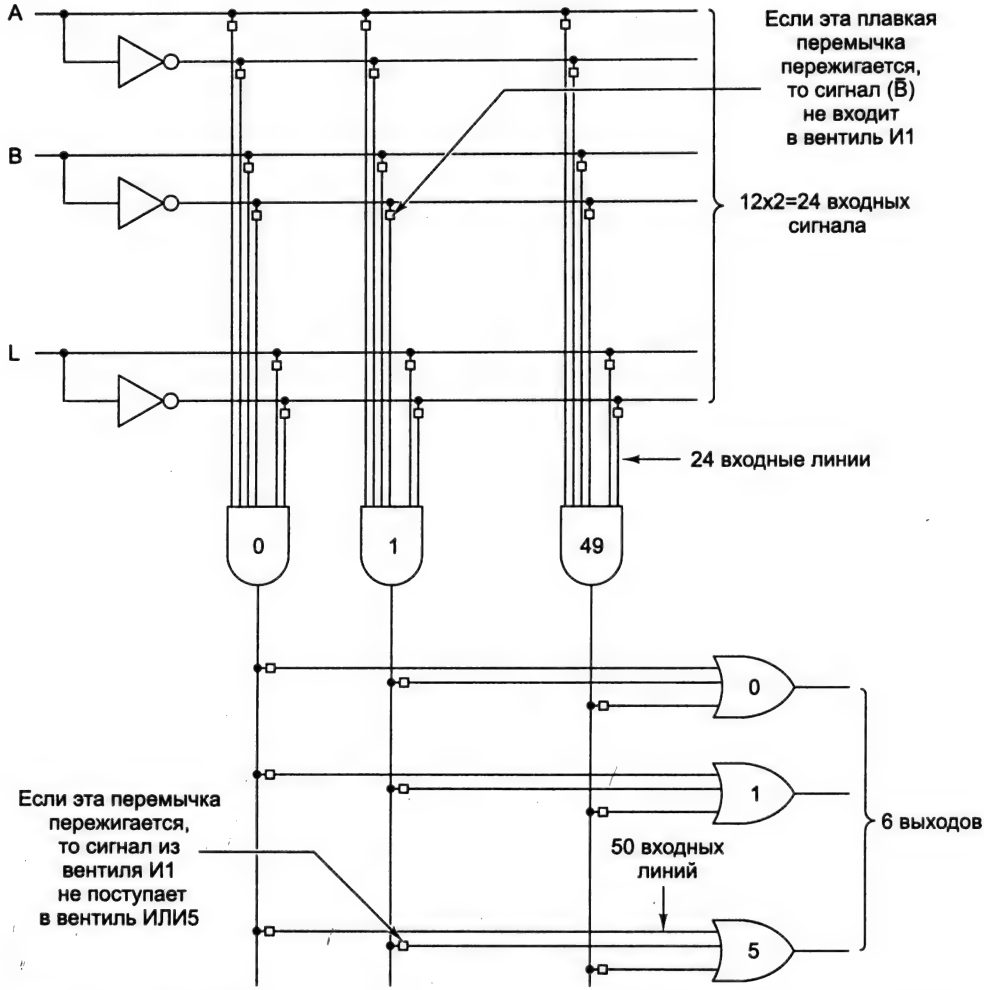


Рис. 3.14. Программируемая логическая матрица с 12 входами и 6 выходами. Маленькие квадратики — плавкие перемычки, выжигаемые для задания функции, которую нужно вычислить. Плавкие перемычки упорядочиваются в двух матрицах. Верхняя матрица — для вентилей И, а нижняя матрица — для вентилей ИЛИ

Матрицы, программируемые в условиях эксплуатации, все еще используются. Однако предпочтение отдается матрицам, которые изготавливаются на заказ. Они разрабатываются заказчиком и выпускаются производителем в соответствии с запросами заказчика. Такие программируемые логические матрицы гораздо дешевле.

А теперь мы можем обсудить три разных способа воплощения таблицы истинности, приведенной на рис. 3.3, *а*. Если в качестве компонентов использовать МИС, нам нужны 4 микросхемы. С другой стороны, мы можем обойтись одним мультиплексором, построенным на СИС, как показано на рис. 3.11, *б*. Наконец, мы можем использовать лишь четвертую часть программируемой логической матрицы. Очевидно, если необходимо вычислять много функций, использование программируе-

мой логической матрицы более эффективно, чем применение двух других методов. Для простых схем предпочтительнее более дешевые МИС и СИС.

Арифметические схемы

Перейдем от СИС общего назначения к комбинационным схемам СИС, которые используются для выполнения арифметических операций. Мы начнем с простой 8-разрядной схемы сдвига, затем рассмотрим структуру сумматоров и, наконец, изучим арифметико-логические устройства, которые играют существенную роль в любом компьютере.

Схемы сдвига

Первой арифметической схемой СИС, которую мы рассмотрим, будет схема сдвига, содержащая 8 входов и 8 выходов (рис. 3.15). Восемь входных битов подаются на линии D_0, ..., D_7. Выходные данные, которые представляют собой входные данные, сдвинутые на 1 бит, поступают на линии S_0, ..., S_7. Линия управления C определяет направление сдвига: 0 — налево, 1 — направо.

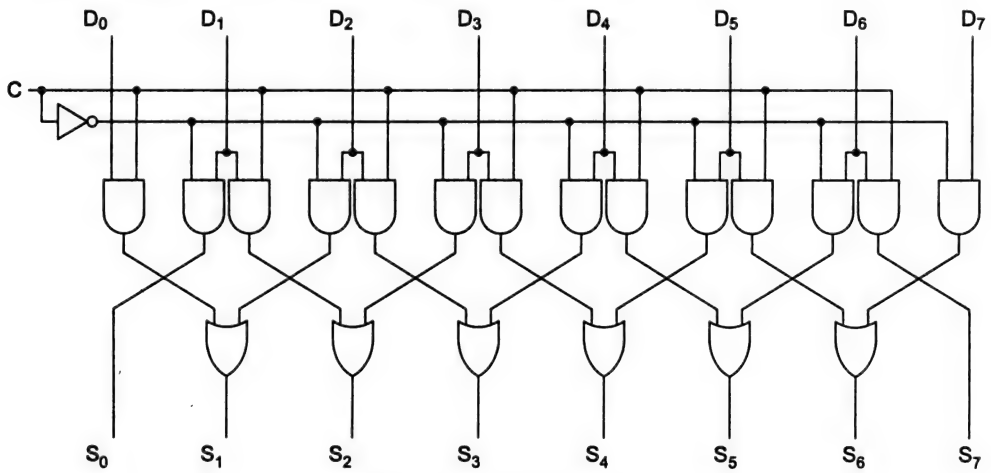


Рис. 3.15. Схема сдвига

Чтобы понять, как работает такая схема, рассмотрим пары вентилей И (кроме крайних вентилей). Если C=1, правый член каждой пары включается, пропуская через себя соответствующий бит. Так как правый вентиль И соединен с входом вентиля ИЛИ, который расположен справа от этого вентиля И, происходит сдвиг вправо. Если C=0, включается левый вентиль И из пары, и тогда происходит сдвиг влево.

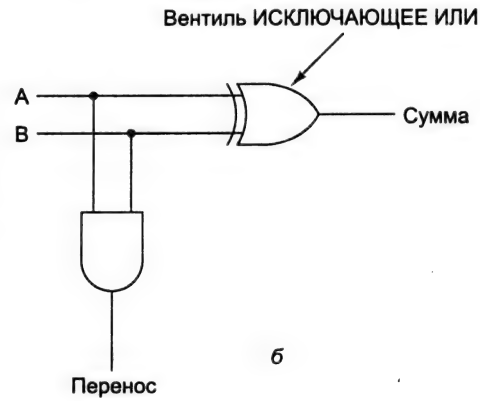
Сумматоры

Компьютер, который не умеет складывать целые числа, практически немыслим. Следовательно, схема для выполнения операций сложения является существенной частью любого процессора. Таблица истинности для сложения одноразряд-

ных целых чисел показана на рис. 3.16, *а*. Здесь имеется два результата: сумма входных переменных A и B и перенос на следующую (левую) позицию. Схема для вычисления бита суммы и бита переноса показана на рис. 3.16, *б*. Такая схема обычно называется **полусумматором**.

A	B	Сумма	Перенос
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

а



б

Рис. 3.16. Таблица истинности для сложения одноразрядных чисел (*а*); схема полусумматора (*б*)

Полусумматор подходит для сложения битов нижних разрядов двух многобитовых слов. Но он не годится для сложения битов в середине слова, потому что не может осуществлять перенос в эту позицию. Поэтому необходим **полный сумматор** (рис. 3.17). Из схемы должно быть ясно, что полный сумматор состоит из двух полусумматоров. Сумма равна 1, если нечетное число переменных A, B и *Вход переноса* принимает значение 1 (то есть если единице равна или одна из переменных, или все три). *Выход переноса* принимает значение 1, если или A и B одновременно равны 1 (левый вход в вентиль ИЛИ), или если один из них равен 1, а *Вход переноса* также равен 1. Два полусумматора порождают и биты суммы, и биты переноса.

Чтобы построить сумматор, например, для двух 16-битных слов, нужно продублировать схему, изображенную на рис. 3.17, *б*, 16 раз. Перенос производится в левый соседний бит. Перенос в самый правый бит соединен с 0. Такой сумматор называется **сумматором со сквозным переносом**. Прибавление 1 к числу 111...111 не осуществится до тех пор, пока перенос не пройдет весь путь от самого правого бита к самому левому. Существуют более быстрые сумматоры, работающие без подобной задержки. Естественно, предпочтение обычно отдается им.

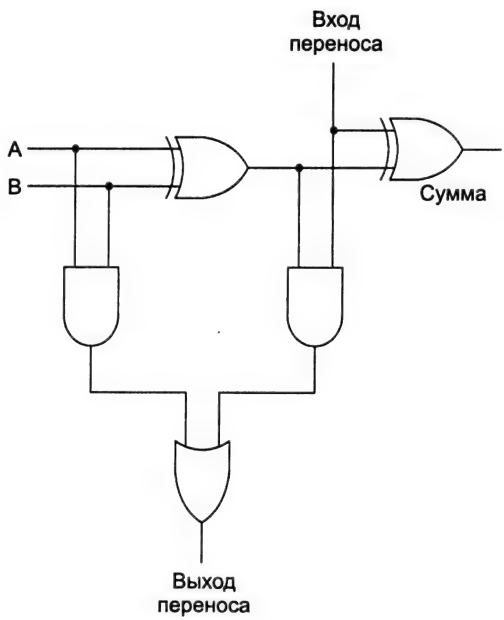
Рассмотрим пример более быстрого сумматора. Разобьем 32-разрядный сумматор на 2 половины: нижнюю 16-разрядную и верхнюю 16-разрядную. Когда начинается сложение, верхний сумматор еще не может приступить к работе, поскольку он не узнает значение переноса, пока не совершится 16 суммирований в нижнем сумматоре.

Однако можно сделать одно преобразование. Вместо одного верхнего сумматора можно получить два верхних сумматора, продублировав соответствующую часть аппаратного обеспечения. Тогда схема будет состоять из трех 16-разрядных сум-

маторов: одного нижнего и двух верхних U0 и U1, которые работают параллельно. В сумматор U0 в качестве переноса поступает 0, а в сумматор U1 в качестве переноса поступает 1. Оба верхних сумматора начинают работу одновременно с нижним сумматором, но только один из результатов суммирования в двух верхних сумматорах будет правильным. После сложения 16 нижних разрядов становится известно значение переноса в верхний сумматор, и тогда можно определить правильный ответ. При таком подходе время сложения сокращается в два раза. Такой сумматор называется **сумматором с выбором переноса**. Можно разбить каждый 16-разрядный сумматор на два 8-разрядных и т. д.

A	B	Вход переноса	Сумма	Выход переноса
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

а



б

Рис. 3.17. Таблица истинности для полного сумматора (*а*); схема для полного сумматора (*б*)

Арифметико-логические устройства

Большинство компьютеров содержат одну схему для выполнения операций И, ИЛИ и сложения над двумя машинными словами. Обычно такая схема для n-битных слов состоит из n идентичных схем для индивидуальных битовых позиций. На рис. 3.18 изображена такая схема, которая называется **арифметико-логическим устройством**, или **АЛУ**. Это устройство может вычислять одну из 4 следующих функций: A И B, A ИЛИ B, $\overline{B}$ и A+B. Выбор функции зависит от того, какие сигналы поступают на линии F_0 и F_1: 00, 01, 10 или 11 (в двоичной системе счисления). Отметим, что здесь A+B означает арифметическую сумму A и B, а не логическую операцию И.

В левом нижнем углу схемы находится двухразрядный декодер, который порождает сигналы включения для четырех операций. Выбор операции определяет-

ся сигналами управления F_0 и F_1. В зависимости от значений F_0 и F_1 выбирается одна из четырех линий разрешения, и тогда выходной сигнал выбранной функции проходит через последний вентиль ИЛИ.

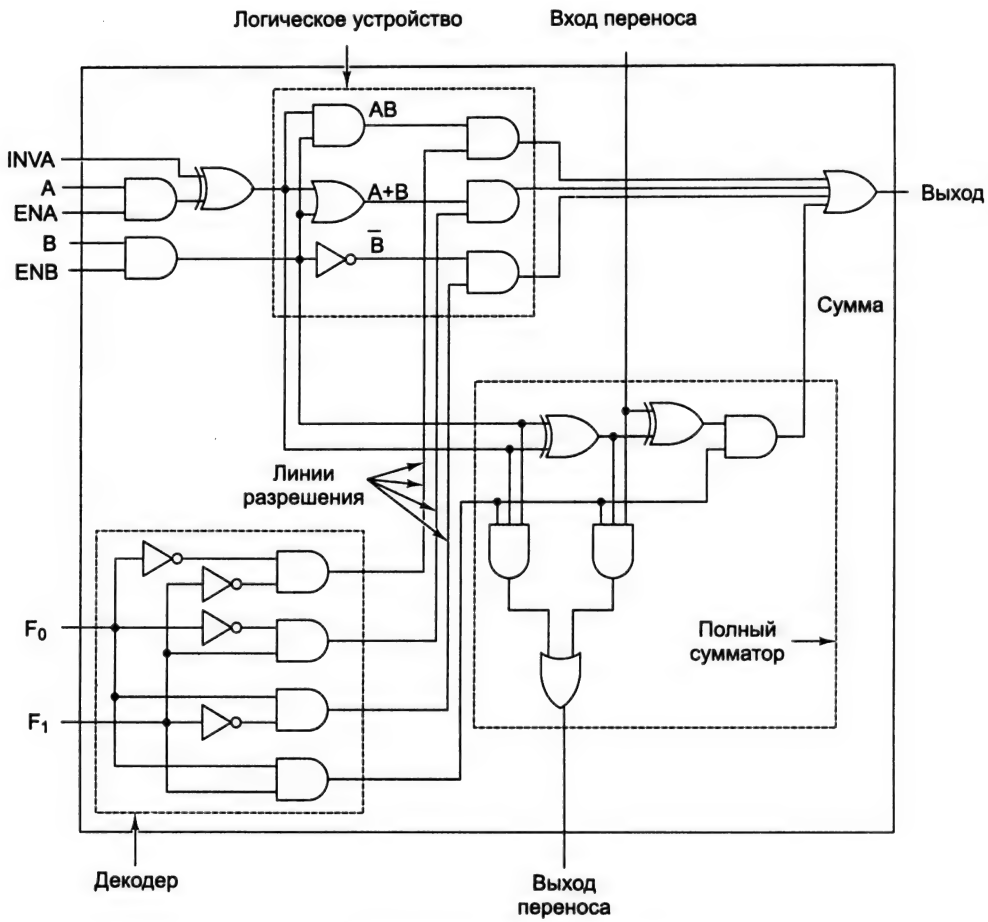


Рис. 3.18. Одноразрядное АЛУ

В верхнем левом углу схемы находится логическое устройство для вычисления A И B, A ИЛИ B и $\overline{B}$, но по крайней мере один из этих результатов проходит через последний вентиль ИЛИ в зависимости от того, какую из разрешающих линий выбрал декодер. Так как ровно один из выходных сигналов декодера будет равен 1, то и запускаться будет ровно один из четырех вентилей И. Остальные три вентиля будут выдавать 0 независимо от значений A и B.

АЛУ может выполнять не только логические и арифметические операции над A и B, но и делать их равными нулю, отрицая ENA (сигнал разрешения A) или ENB (сигнал разрешения B). Можно также получить $\overline{A}$, установив INVA (инверсию A). Зачем нужны ENA, ENB и INVA, мы рассмотрим в главе 4. При нормаль-

ных условиях и ENA, и ENB равны 1, чтобы разрешить поступление обоих входных сигналов, а сигнал INVA равен 0. В этом случае A и B просто поступают в логическое устройство без изменений.

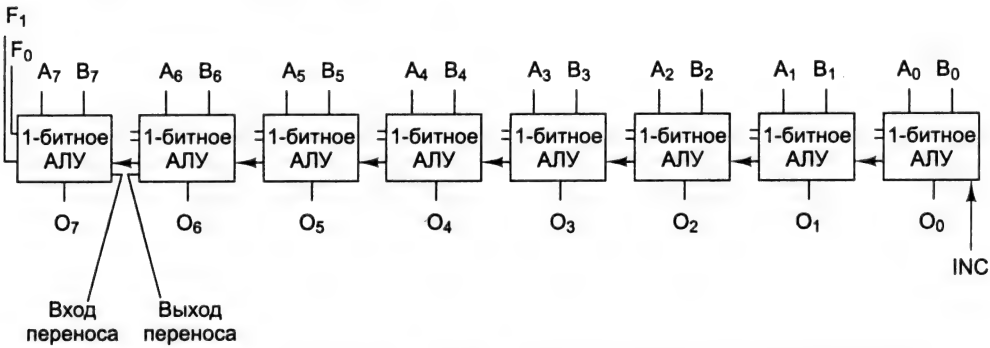


Рис. 3.19. Восемь одноразрядных секций, соединенных в 8-разрядное АЛУ. Сигналы разрешения и инверсии не показаны для упрощения схемы

В нижнем правом углу находится полный сумматор для подсчета суммы A и B и для осуществления переносов. Переносы необходимы, поскольку несколько таких схем могут быть соединены для выполнения операций над целыми словами. Одноразрядные схемы, подобные той, которая изображена на рис. 3.18, называются разрядными микропроцессорными секциями. Они позволяют разработчику сконструировать АЛУ любой желаемой ширины. На рис. 3.19 показана схема 8-разрядного АЛУ, составленного из восьми **одноразрядных секций**. Сигнал INC (увеличение на единицу) нужен только для операций сложения. Он дает возможность вычислять такие суммы, как A+1 и A+B+1.

Тактовые генераторы

Во многих цифровых схемах все зависит от порядка, в котором выполняются действия. Иногда одно действие должно предшествовать другому, иногда два действия должны происходить одновременно. Для контроля временных отношений в цифровые схемы встраиваются тактовые генераторы, чтобы обеспечить синхронизацию. **Тактовый генератор** — это схема, которая вызывает серию импульсов. Все импульсы одинаковы по длительности. Интервалы между последовательными импульсами также одинаковы. Временной интервал между началом одного импульса и началом следующего называется **временем такта**. Частота импульсов обычно от 1 до 500 МГц, что соответствует времени такта от 1000 нс до 2 нс. Частота тактового генератора обычно контролируется кварцевым генератором, чтобы достичь высокой точности.

В компьютере за время одного такта может произойти много событий. Если они должны осуществляться в определенном порядке, то такт следует разделить на подтакты. Чтобы достичь лучшего разрешения, чем у основного тактового генератора, нужно сделать ответвление от задающей линии тактового генератора и вставить схему с определенным временем задержки. Таким образом порождается

вторичный сигнал тактового генератора, который сдвинут по фазе относительно первичного (рис. 3.20, *а*). Временная диаграмма (рис. 3.20, *б*) обеспечивает четыре начала отсчета времени для дискретных событий:

1. Нарастающий фронт С1.
2. Задний фронт С1.
3. Нарастающий фронт С2.
4. Задний фронт С2.

Связав различные события с различными фронтами, можно достичь требуемой последовательности выполнения действий. Если в пределах одного такта требуется более четырех начал отсчета, можно сделать еще несколько ответвлений от задающей линии с различным временем задержки.

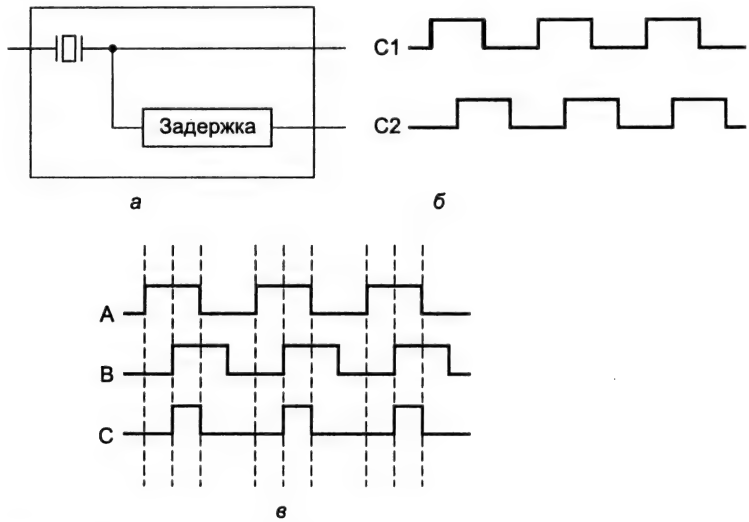


Рис. 3.20. Тактовый генератор (*а*); временная диаграмма для тактового генератора (*б*); порождение асинхронных тактовых импульсов (*в*)

В некоторых схемах важны временные интервалы, а не дискретные моменты времени. Например, некоторое событие может происходить в любое время, когда уровень импульса С1 высокий, а не на нарастающем фронте. Другое событие может происходить только в том случае, когда уровень импульса С2 высокий. Если необходимо более двух интервалов, нужно обеспечить больше линий передачи синхронизирующих импульсов или сделать так, чтобы состояния с высоким уровнем импульса у двух тактовых генераторов частично пересекались во времени. В последнем случае можно выделить 4 отдельных интервала: $\overline{C1}$ И $\overline{C2}$, С1 И С2, С1 И $\overline{C2}$ и $\overline{C1}$ И С2.

Тактовые генераторы могут быть синхронными. В этом случае время состояния с высоким уровнем импульса равно времени состояния с низким уровнем импульса (рис. 3.20, *б*). Чтобы получить асинхронную серию импульсов, нужно сдвинуть сигнал задающего генератора, используя цепь задержки. Затем нужно

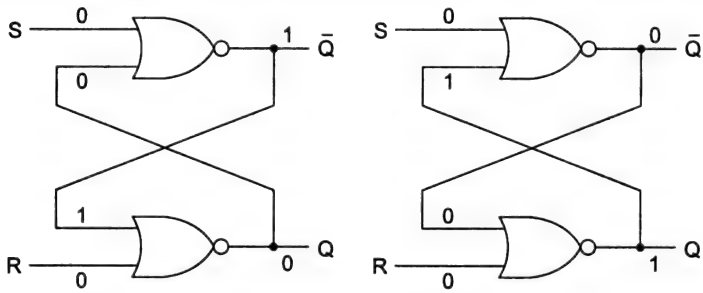
соединить полученный сигнал с изначальным сигналом с помощью логической функции И (см. рис. 3.20, *в*, сигнал C).

Память

Память является необходимым компонентом любого компьютера. Без памяти не было бы компьютеров, по крайней мере таких, какие есть сейчас. Память используется как для хранения команд, которые нужно выполнить, так и данных. В следующих разделах мы рассмотрим основные компоненты памяти, начиная с уровня вентилей. Мы увидим, как они работают и как из них можно получить память большой емкости.

Защелки

Чтобы создать один бит памяти, нам нужна схема, которая каким-то образом «запоминает» предыдущие входные значения. Такую схему можно сконструировать из двух вентилей НЕ-ИЛИ, как показано на рис. 3.21, *а*. Аналогичные схемы можно построить из вентилей НЕ-И. Мы не будем упоминать эти схемы в дальнейшем, поскольку они, по существу, идентичны схемам с вентилями НЕ-ИЛИ.



A	B	НЕ-ИЛИ
0	0	1
0	1	0
1	0	0
1	1	0

в

Рис. 3.21. Защелка НЕ-ИЛИ в состоянии 0 (*а*); защелка НЕ-ИЛИ в состоянии 1 (*б*); таблица истинности для функции НЕ-ИЛИ (*в*)

Схема, изображенная на рис. 3.21, *а*, называется **SR-защелкой**. У нее есть два входа: S (setting — установка) и R (resetting — сброс). У нее также есть два комплементарных[1] (дополнительных) выхода: Q и $\overline{Q}$. В отличие от комбинационной схемы, выходные сигналы защелки не определяются текущими входными сигналами.

Чтобы увидеть, как это осуществляется, предположим, что S=0 и R=0 (вообще они равны 0 большую часть времени). Чтобы провести доказательство, предположим также, что Q=0. Так как Q возвращается в верхний вентиль НЕ-ИЛИ и оба входа этого вентиля равны 0, то его выход, $\overline{Q}$, равен 1. Единица возвращается в нижний вентиль, у которого в итоге один вход равен 0, а другой — 1, а на выходе получается Q=0. Такое положение вещей, по крайней мере, состоятельно (рис. 3.21, *а*).

[1] От англ. *complementary* — дополняющий. — *Примеч. пер.*

А теперь давайте представим, что Q=1, а R и S все еще равны 0. Верхний вентиль имеет входы 0 и 1 и выход $\overline{Q}$ (то есть 0), который возвращается в нижний вентиль. Такое положение вещей, изображенное на рис. 3.21, *б*, также состоятельно. Положение, когда оба выхода равны 0, несостоятельно, поскольку в этом случае оба вентиля имели бы на входе два нуля, что привело бы к единице на выходе, а не к нулю. Точно так же невозможно иметь оба выхода равных 1, поскольку это привело бы к входным сигналам 0 и 1, что вызывает на выходе 0, а не 1. Наш вывод прост: при R=S=0 защелка имеет два стабильных состояния, которые мы будем называть 0 и 1 в зависимости от Q.

А сейчас давайте рассмотрим действие входных сигналов на состояние защелки. Предположим, что S принимает значение 1, в то время как Q=0. Тогда входные сигналы верхнего вентиля будут 1 и 0, что приведет к выходному сигналу $\overline{Q}$=0. Это изменение делает оба входа в нижний вентиль равными 0 и, следовательно, выходной сигнал равным 1. Таким образом, установка S на значение 1 переключает состояние с 0 на 1. Установка R на значение 1, когда защелка находится в состоянии 0, не вызывается изменений, поскольку выход нижнего вентиля НЕ-ИЛИ равен 0 и для входов 10, и для входов 11.

Используя подобную аргументацию, легко увидеть, что установка S на значение 1 при состоянии защелки 1 (то есть при Q=1) не вызывает изменений, но установка R на значение 1 приводит к изменению состояния защелки. Таким образом, если S принимает значение 1, то Q будет равно 1 независимо от предыдущего состояния защелки. Сходным образом переход R на значение 1 вызывает Q=0. Схема «запоминает», какой сигнал был в последний раз: S или R. Используя это свойство, мы можем конструировать компьютерную память.

Синхронные SR-защелки

Часто бывает удобно сделать так, чтобы защелка меняла состояние только в определенные моменты. Чтобы достичь этой цели, мы немного изменили основную схему и получили **синхронную SR-защелку** (рис. 3.22).

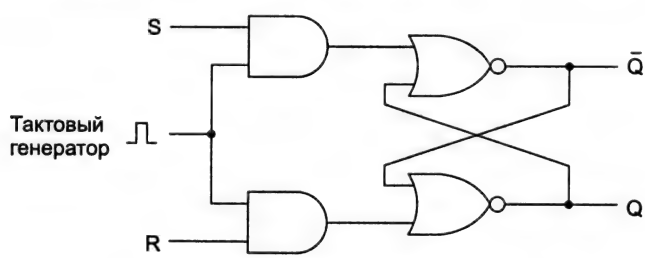


Рис. 3.22. Синхронная SR-защелка

Эта схема имеет дополнительный синхронизирующий вход, который обычно равен 0. Если этот вход равен 0, то оба выхода вентилей И равны 0 независимо от S и R, и защелка не меняет состояние. Когда значение синхронизирующего входа равно 1, действие вентилей И исчезает и состояние защелки становится зависимым от S и R. Для обозначения того факта, что синхронизирующий вход равен 1

(то есть состояние схемы зависит от значений S и R), часто используется термин **стробировать**.

До сих пор мы скрывали, что происходит, если S=R=1. И по понятным причинам: когда и R, и S в конце концов возвращаются к 0, схема становится недетерминированной. Единственное состоятельное положение при S=R=1 — это $Q=\overline{Q}=0$, но как только оба входа возвращаются к 0, защелка должна перейти в одно из двух стабильных состояний. Если один из входов принимает значение 0 раньше, чем другой, оставшийся в состоянии 1 «побеждает», потому что когда один из входов равен 1, он управляет состоянием защелки. Если оба входа переходят к 0 одновременно (что маловероятно), защелка переходит в одно из своих состояний наугад.

Синхронные D-защелки

Чтобы разрешить неопределенность SR-защелки (неопределенность возникает в случае, если S=R=1), нужно предотвратить появление подобной неопределенности. На рис. 3.23 изображена схема защелки только с одним входом D. Так как входной сигнал в нижний вентиль И всегда является обратным кодом входного сигнала в верхний вентиль И, ситуация, когда оба входа равны 1, никогда не возникает. Когда D=1 и синхронизирующий вход равен 1, защелка переходит в состояние Q=1. Когда D=0 и синхронизирующий вход равен 1, защелка переходит в состояние Q=0. Другими словами, когда синхронизирующий вход равен 1, текущее значение D отбирается и сохраняется в защелке. Такая схема, которая называется **синхронной D-защелкой**, представляет собой память объемом 1 бит. Значение, которое было сохранено, всегда доступно на выходе Q. Чтобы загрузить в память текущее значение D, нужно пустить положительный импульс по линии синхронизирующего сигнала.

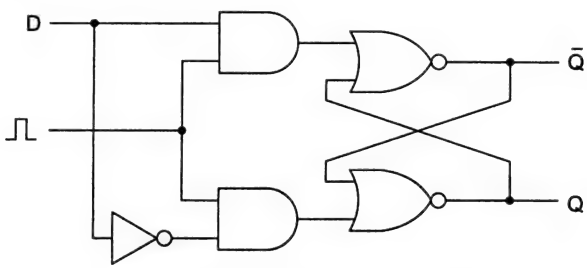


Рис. 3.23. Синхронная D-защелка

Такая схема требует наличия 11 транзисторов. Более сложные схемы могут хранить 1 бит, имея всего 6 транзисторов. На практике обычно используются последние.

Триггеры (flip-flops)

Многие схемы выбирают значение на определенной линии в определенный момент времени и запоминают его. В такой схеме, которая называется **триггером**,

переход состояния происходит не тогда, когда синхронизирующий сигнал равен 1, а во время перехода синхронизирующего сигнала с 0 на 1 (нарастающий фронт) или с 1 на 0 (задний фронт). Следовательно, длина синхронизирующего импульса не имеет значения, поскольку переходы происходят быстро.

Подчеркнем еще раз различие между триггером и защелкой. Триггер **запускается фронтом сигнала**, а защелка **запускается уровнем сигнала**. Обратите внимание, что в литературе эти термины часто путаются. Многие авторы используют термин «триггер», когда речь идет о защелке, и наоборот[1].

Существует несколько подходов к разработке триггеров. Например, если бы существовал способ генерирования очень короткого импульса на нарастающем фронте синхронизирующего сигнала, этот импульс можно было бы подавать в D-защелку. В действительности такой способ существует. Соответствующая схема показана на рис. 3.24, *а*.

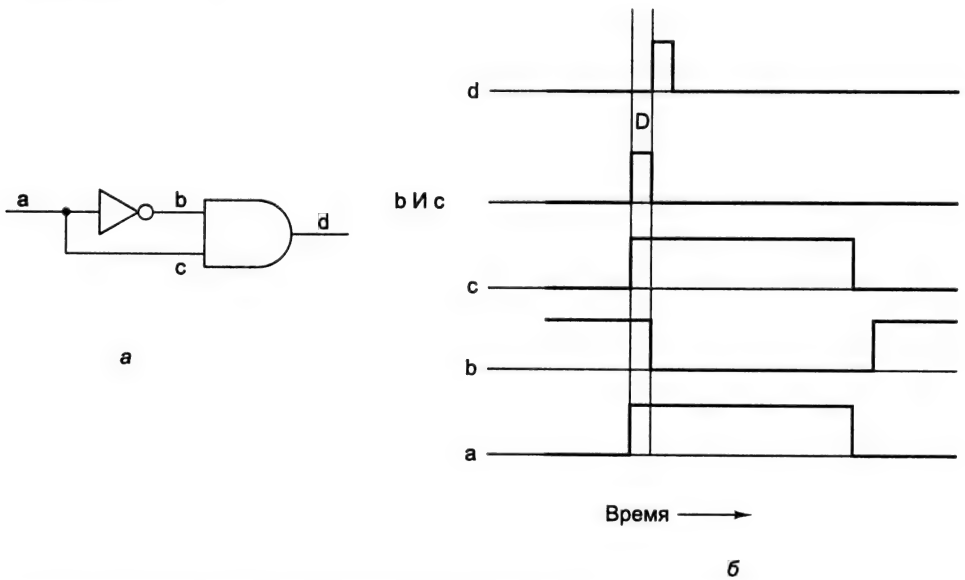


Рис. 3.24. Генератор импульса (*а*); временная диаграмма для четырех точек на схеме (*б*)

На первый взгляд может показаться, что выход вентиля И всегда будет нулевым, поскольку функция И от любого сигнала с его инверсией дает 0, но на самом деле ситуация несколько более тонкая. При прохождении сигнала через инвертор происходит небольшая, но все-таки не нулевая задержка. Данная схема работает именно благодаря этой задержке. Предположим, что мы измеряем напряжение в четырех точках a, b, c и d. Входовый сигнал в точке a представляет собой длинный синхронизирующий импульс (см. нижний график на рис. 3.24, *б*). Сигнал в точке b показан над ним. Отметим, что этот сигнал инвертирован и подается с некоторой

[1] В отечественной литературе термин «защелка» (latch) не используется, говорят о триггерах. Однако при этом вводится понятие Т-триггера, который здесь называется настоящим триггером. — *Примеч. научн. ред.*

задержкой. Время задержки зависит от типа инвертора и обычно составляет несколько наносекунд.

Сигнал в точке с тоже подается с задержкой, но эта задержка обусловлена только временем прохождения сигнала (со скоростью света). Если физическое расстояние между a и c, например, 20 микрон, тогда задержка на распространение сигнала составляет 0,0001 нс, что, конечно, незначительно по сравнению со временем, которое требуется на прохождение сигнала через инвертор. Таким образом, сигнал в точке с практически идентичен сигналу в точке a.

Когда входные сигналы b и c подвергаются операции И, в результате получается короткий импульс, длина которого (Δ) равна вентильной задержке инвертора (обычно 5 нс и меньше). Выходной сигнал вентиля И — данный импульс, сдвинутый из-за задержки вентиля И (см. верхний график на рис. 3.24, *б*). Этот временной сдвиг означает только то, что D-защелка активизируется с определенной задержкой после нарастающего фронта синхронизирующего импульса. Он никак не влияет на длину импульса. В памяти со временем цикла в 50 нс импульс в 5 нс (который сообщает, когда нужно выбирать линию D) достаточно короткий, и в этом случае полная схема может быть такой, какая изображена на рис. 3.25. Следует упомянуть, что такая схема триггера проста для понимания, но на практике обычно используются более сложные триггеры.

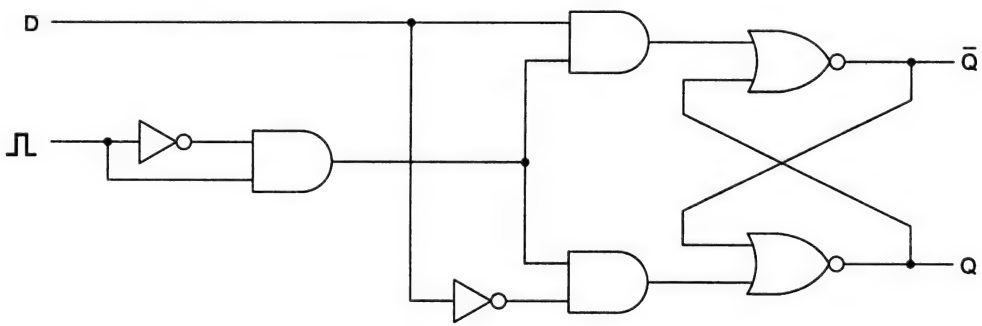


Рис. 3.25. D-триггер

Стандартные изображения защелок и триггеров показаны на рис. 3.26. На рис. 3.26, *а* изображена защелка, состояние которой загружается тогда, когда синхронизирующий сигнал CK (от слова clock) равен 1, в противоположность защелке, изображенной на рис. 3.26, *б*, у которой синхронизирующий сигнал обычно равен 1, но переходит на 0, чтобы загрузить состояние из D. На рис. 3.26, *в* и *г* изображены триггеры. То, что это триггеры, а не защелки, показано с помощью уголка при синхронизирующем входе. Триггер на рис. 3.26, *в* изменяет состояние на возрастающем фронте синхронизирующего импульса (переход от 0 к 1), тогда как триггер на рис. 3.26, *г* изменяет состояние на заднем фронте (переход от 0 к 1). Многие (хотя не все) защелки и триггеры также имеют выход $\overline{Q}$, а у некоторых есть два дополнительных входа: *Set* (установка) или *Preset* (предварительная установка) и *Reset* (сброс) или *Clear* (очистка). Первый вход (*Set* или *Preset*) устанавливает Q=1, а второй (*Reset* или *Clear*) — Q=0.

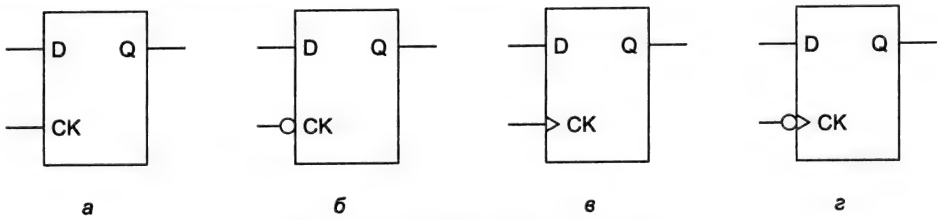


Рис. 3.26. D-защелки и D-триггеры

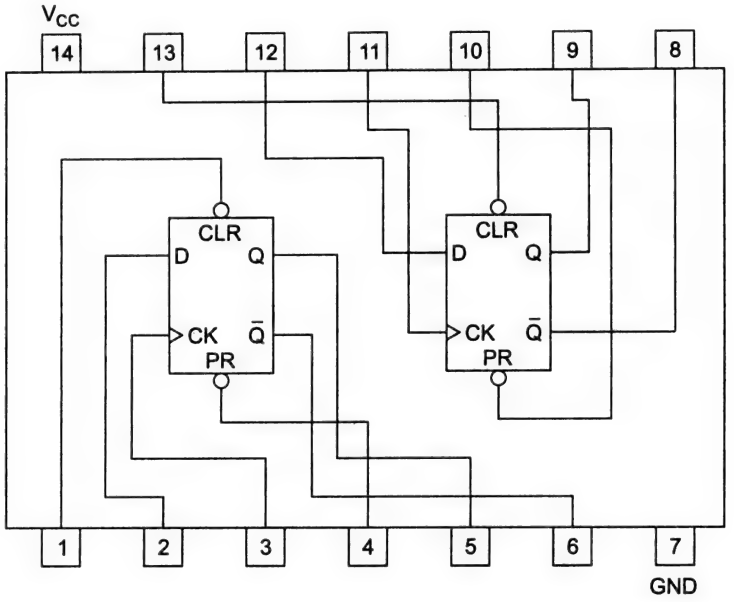
Регистры

Существуют различные конфигурации триггеров. На рисунке 3.27, *а* изображена схема, содержащая два независимых D-триггера с сигналами предварительной установки и очистки. Хотя эти два триггера находятся на одной микросхеме с 14 выводами, они не связаны между собой. Совершенно по-другому устроен восьмиразрядный триггер, изображенный на рис. 3.27, *б*. Здесь, в отличие от предыдущей схемы, у восьми триггеров нет выхода $\overline{Q}$ и линий предварительной установки и все синхронизирующие линии связаны вместе и управляются выводом 11. Сами триггеры того же типа, что на рис. 3.26, *г*, но инвертирующие входы аннулируются инвертором, связанным с выводом 11, поэтому триггеры запускаются при переходе от 0 к 1. Все восемь сигналов очистки также объединены, поэтому когда вывод 1 переходит в состояние 0, все триггеры также переходят в состояние 0. Если вам не понятно, почему вывод 11 инвертируется на входе, а затем инвертируется снова при каждом сигнале CK, то ответ прост: входной сигнал не имеет достаточной мощности, чтобы запустить все восемь триггеров; входной инвертор на самом деле используется в качестве усилителя.

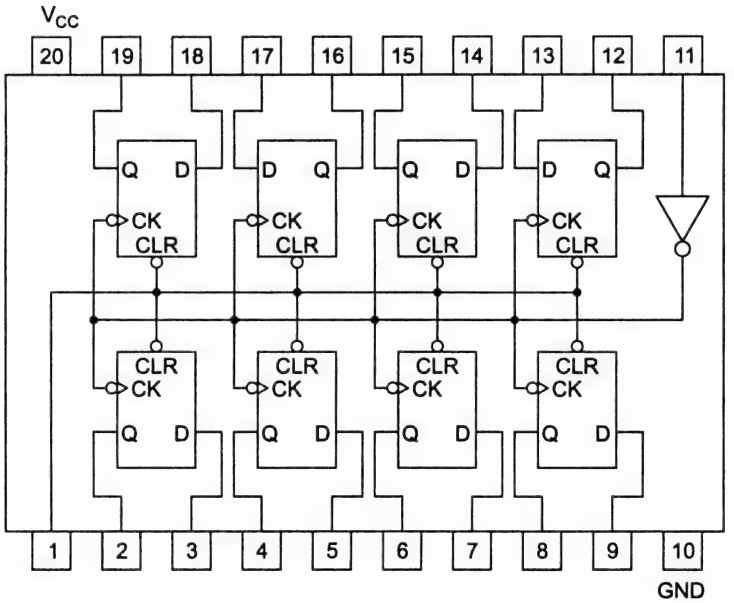
Одна из причин объединения линий синхронизации и линий очистки в микросхеме на рис. 3.27, *б* — экономия выводов. С другой стороны, микросхема данной конфигурации несколько отличается от восьми несвязанных триггеров. Эта микросхема используется в качестве одного 8-разрядного регистра. Две такие микросхемы могут работать параллельно, образуя 16-разрядный регистр. Для этого нужно связать соответствующие выводы 1 и 11. Регистры и их применение мы рассмотрим более подробно в главе 4.

Организация памяти

Хотя мы и совершили переход от простой памяти в 1 бит (см. рис. 3.23) к 8-разрядной памяти (см. рис. 3.27, *б*), чтобы построить память большого объема, требуется другой способ организации, при котором можно обращаться к отдельным словам. Пример организации памяти, которая удовлетворяет этому критерию, показан на рис. 3.28. Эта память содержит четыре 3-битных слова. Каждая операция считывает или записывает целое 3-битное слово. Хотя общий объем памяти (12 битов) ненамного больше, чем у нашего 8-разрядного триггера, такая память требует меньшего количества выводов, и, что особенно важно, подобная организация применима при построении памяти большого объема.



а



б

Рис. 3.27. Два D-триггера (*а*); восьмиразрядный триггер (*б*)

Хотя структура памяти, изображенная на рис. 3.28, может на первый взгляд показаться сложной, на самом деле она очень проста благодаря своей регулярной структуре. Она содержит 8 входных линий (3 входа для данных — I_0, I_1 и I_2; 2 входа

для адресов — A_0 и A_1; 3 входа для управления — CS (Chip Select — выбор элемента памяти), RD (для различия между считыванием и записью) и OE (Output Enable — разрешение выдачи выходных сигналов)) и 3 выходные линии для данных — O_0, O_1 и O_2. Такую память в принципе можно поместить в корпус с 14 выводами (включая питание и «землю»), а 8-разрядный триггер требует наличия 20 выводов.

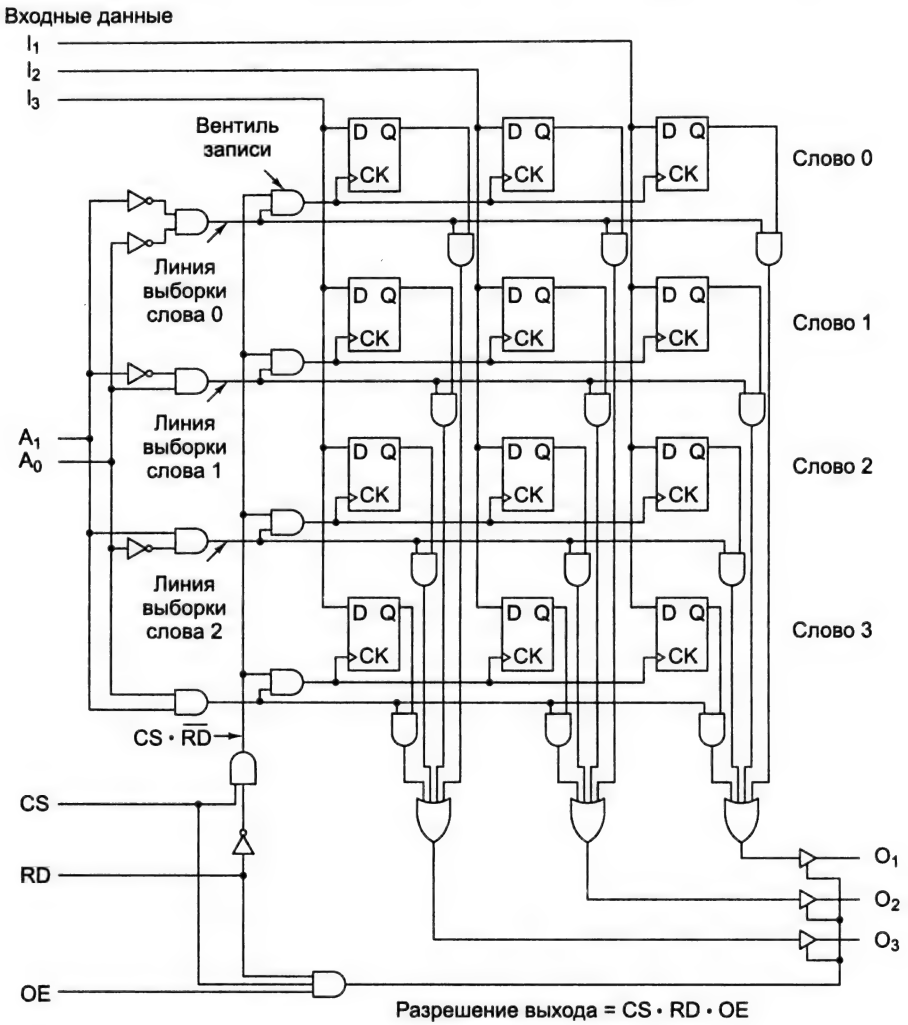


Рис. 3.28. Логическая блок-схема для памяти 4×3. Каждый ряд представляет одно из 3-битных слов. При операции считывания и записи всегда считывается или записывается целое слово

Чтобы выбрать микросхему памяти, внешняя логика должна установить CS на 1, а также установить RD на 1 для чтения и на 0 для записи. Две адресные линии должны указывать, какое из четырех 3-битных слов нужно считывать или записывать. При операции считывания входные линии для данных не используются.

Выбирается слово и помещается на выходные линии для данных. При операции записи биты, находящиеся на входных линиях для данных, загружаются в выбранное слово памяти; выходные линии при этом не используются.

А теперь давайте посмотрим, как работает память, изображенная на рис. 3.28. Четыре вентиля И для выбора слов в левой части схемы формируют декодер. Входные инверторы расположены так, что каждый вентиль запускается определенным адресом. Каждый вентиль приводит в действие линию выбора слов (для слов 0, 1, 2 и 3). Когда микросхема должна производить запись, вертикальная линия $CS \cdot \overline{RD}$ получает значение 1, запуская один из 4 вентилей записи. Выбор вентиля зависит от того, какая именно линия выбора слов равна 1. Выходной сигнал вентиля записи приводит в действие все сигналы CK для выбранного слова, загружая входные данные в триггеры для этого слова. Запись производится только в том случае, если CS равно 1, а RD равно 0, при этом записывается только слово, выбранное адресами A_0 и A_1; остальные слова не меняются.

Процесс считывания сходен с процессом записи. Декодирование адреса происходит точно так же, как и при записи. Но в данном случае линия $CS \cdot \overline{RD}$ принимает значение 0, поэтому все вентили записи блокируются и ни один из триггеров не меняется. Вместо этого линия выбора слов запускает вентили И, связанные с битами Q выбранного слова. Таким образом, выбранное слово передает свои данные в четырехвходовые вентили ИЛИ, расположенные в нижней части схемы, а остальные три слова выдают 0. Следовательно, выход вентилей ИЛИ идентичен значению, сохраненному в данном слове. Остальные три слова никак не влияют на выходные данные.

Мы могли бы разработать схему, в которой три вентиля ИЛИ соединялись бы с тремя линиями вывода данных, но это вызвало бы некоторые проблемы. Мы рассматривали линии ввода данных и линии вывода данных как разные линии. На практике же используются одни и те же линии. Если бы мы связали вентили ИЛИ с линиями вывода данных, микросхема пыталась бы выводить данные (то есть задавать каждой линии определенную величину) даже в процессе записи, мешая нормальному вводу данных. По этой причине желательно каким-то образом соединять вентили ИЛИ с линиями вывода данных при считывании и полностью разъединять их при записи. Все, что нам нужно, — электронный переключатель, который может устанавливать и разрушать связь за несколько наносекунд.

К счастью, такие переключатели существуют. На рис. 3.29, *а* показано символическое изображение так называемого **буферного элемента без инверсии**. Он содержит вход для данных, выход для данных и вход управления. Когда вход управления равен 1, буферный элемент работает как провод (см. рис. 3.29, *б*). Когда вход управления равен 0, буферный элемент работает как разомкнутая цепь (см. рис. 3.29, *в*), как будто кто-то отрезал выход для данных от остальной части схемы кусачками. Соединение может быть восстановлено за несколько наносекунд, если сделать сигнал управления равным 1.

На рис. 3.29, *г* показан **буферный элемент с инверсией**, который действует как обычный инвертор, когда сигнал управления равен 1, и отделяет выход от остальной части схемы, когда сигнал управления равен 0. Оба буферных элемента представляют собой **устройства с тремя состояниями**, поскольку они могут выдавать 0, 1 или вообще не выдавать сигнала (в случае с разомкнутой цепью). Буфер-

ные элементы, кроме того, усиливают сигналы, поэтому они могут справляться с большим количеством сигналов одновременно. Иногда они используются в схемах именно по этой причине, даже если их свойства переключателя не нужны.

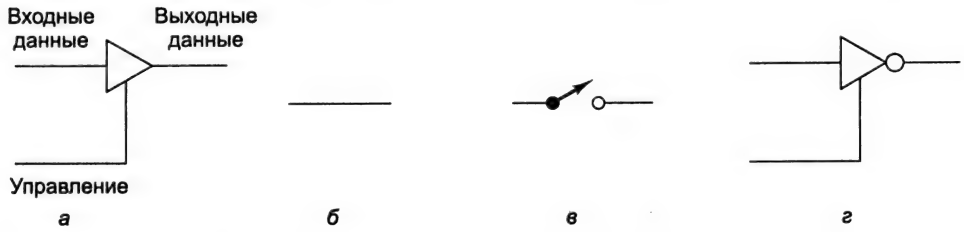


Рис. 3.29. Буферный элемент без инверсии (*а*); действие буферного элемента без инверсии, когда сигнал управления равен 1 (*б*); действие буферного элемента без инверсии, когда сигнал управления равен 0 (*в*); буферный элемент с инверсией (*г*)

Сейчас уже должно быть понятно, для чего нужны три буферных элемента без инверсии на линиях вывода данных. Когда CS, RD и OE все равны 1, то сигнал разрешения выдачи выходных данных также равен 1, в результате чего запускаются буферные элементы и слово помещается на выходные линии. Когда один из сигналов CS, RD и OE равен 0, выходы отсоединяются от остальной части схемы.

Микросхемы памяти

Преимущество памяти, изображенной на рис. 3.28, состоит в том, что подобная структура применима при разработке памяти большого объема. Мы нарисовали схему 4×3 (для 4 слов по 3 бита каждое). Чтобы расширить ее до размеров 4×8, нужно добавить еще 5 колонок триггеров по 4 триггера в каждой, а также 5 входных и 5 выходных линий. Чтобы перейти от размера 4×3 к размеру 8×3, мы должны добавить еще четыре ряда триггеров по три триггера в каждом, а также адресную линию A_2. При такой структуре число слов в памяти должно быть степенью двойки для максимальной эффективности, а число битов в слове может быть любым.

Поскольку технология изготовления интегральных схем хорошо подходит для производства микросхем с внутренней структурой повторяемой плоской поверхности, микросхемы памяти являются идеальным применением для этого. С развитием технологии число битов, которое можно вместить в одной микросхеме, постоянно увеличивается, обычно в два раза каждые 18 месяцев (закон Мура). С появлением больших микросхем маленькие микросхемы не всегда устаревают из-за компромиссов между преимуществами емкости, скорости, мощности, цены и сопряжения. Обычно самые большие современные микросхемы пользуются огромным спросом и, следовательно, стоят гораздо дороже за 1 бит, чем микросхемы небольшого размера.

При любом объеме памяти существует несколько различных способов организации микросхемы. На рис. 3.30 показаны две возможные структуры микросхемы в 4 Мбит: 512 K×8 и 4096 K×1. (Размеры микросхем памяти обычно даются в битах, а не в байтах, поэтому здесь мы будем придерживаться этого соглашения.) На рис. 3.30, *а* можно видеть 19 адресных линий для обращения к одному из 2^{19} байтов и 8 линий данных для загрузки или хранения выбранного байта.

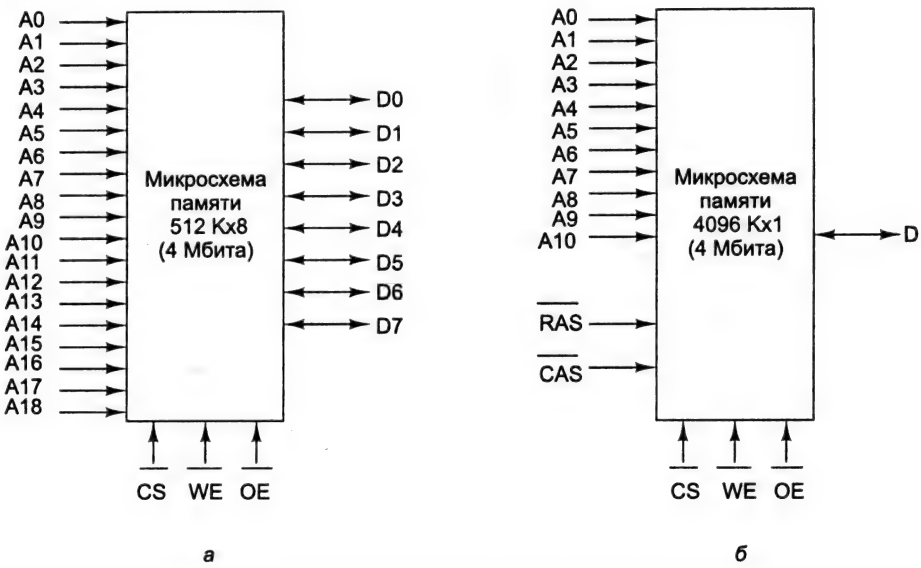


Рис. 3.30. Два способа организации памяти объемом 4 Мбит

Сделаем небольшое замечание по поводу терминологии. На одних выводах высокое напряжение вызывает какое-либо действие, на других — низкое напряжение. Чтобы избежать путаницы, мы будем употреблять термин **«установить сигнал»**, когда вызывается какое-то действие, вместо того чтобы говорить, что напряжение повышается или понижается. Таким образом, для одних выводов установка сигнала значит установку на 1, а для других — установку на 0. Названия выводов, которые устанавливаются на 0, содержат сверху черту. Сигнал CS устанавливается на 1, а сигнал $\overline{\text{CS}}$ — на 0. Противоположный термин — **«сбросить»**.

А теперь вернемся к нашей микросхеме. Поскольку обычно компьютер содержит много микросхем памяти, нужен сигнал для выбора необходимой микросхемы, такой, чтобы нужная нам микросхема реагировала на вызов, а остальные нет. Сигнал $\overline{\text{CS}}$ (Chip Select — выбор элемента памяти) используется именно для этой цели. Он устанавливается, чтобы запустить микросхему. Кроме того, нужен способ отличия считывания от записи. Сигнал $\overline{\text{WE}}$ (Write Enable — разрешение записи) используется для указания того, что данные должны записываться, а не считываться. Наконец, сигнал $\overline{\text{OE}}$ (Output Enable — разрешение выдачи выходных сигналов) устанавливается для выдачи выходных сигналов. Когда этого сигнала нет, выход отсоединен от остальной части схемы.

На рис. 3.30, *б* используется другая схема адресации. Микросхема представляет собой матрицу 2048×2048 однобитных ячеек, что составляет 4 Мбит. Чтобы обратиться к микросхеме, сначала нужно выбрать строку. Для этого 11-битный номер этой строки подается на адресные выводы. Затем устанавливается сигнал $\overline{\text{RAS}}$ (Row Address Strobe — строб адреса строки). После этого на адресные выводы подается номер столбца и устанавливается сигнал $\overline{\text{CAS}}$ (Column Address Strobe — строб адреса столбца). Микросхема реагирует на сигнал, принимая или выдавая 1 бит данных.

Большие микросхемы памяти часто производятся в виде матриц m×n, обращение к которым происходит по строке и столбцу. Такая организация памяти сокращает число необходимых выводов, но, с другой стороны, замедляет обращение к микросхеме, поскольку требуется два цикла адресации: один для строки, а другой для столбца. Чтобы ускорить этот процесс, в некоторых микросхемах можно вызывать адрес ряда, а затем несколько адресов столбцов для доступа к последовательным битам ряда.

Много лет назад самые большие микросхемы памяти обычно были устроены так, как показано на рис. 3.30, *б*. Поскольку слова выросли от 8 до 32 битов и выше, использовать подобные микросхемы стало неудобно. Чтобы из микросхем 4096 K×1 построить память с 32-битными словами, требуется 32 микросхемы, работающие параллельно. Эти 32 микросхемы имеют общий объем, по крайней мере, 16 Мбайт. Если использовать микросхемы 512 K×8, то потребуется всего 4 микросхемы, но при этом объем памяти будет составлять 2 Мбайт. Чтобы избежать наличия 32 микросхем, большинство производителей выпускают семейства микросхем с длиной слов 1, 4, 8 и 16 битов.

ОЗУ и ПЗУ

Все виды памяти, которые мы рассматривали до сих пор, имеют одно общее свойство: в них можно и записывать информацию, и считывать ее. Такая память называется **ОЗУ (оперативное запоминающее устройство)**. Существует два типа ОЗУ: статическое и динамическое. **Статическое ОЗУ** конструируется с использованием D-триггеров. Информация в ОЗУ сохраняется на протяжении всего времени, пока к нему подается питание: секунды, минуты, часы и даже дни. Статическое ОЗУ работает очень быстро. Обычно время доступа составляет несколько наносекунд. По этой причине статическое ОЗУ часто используется в качестве кэш-памяти второго уровня.

В **динамическом ОЗУ**, напротив, триггеры не используются. Динамическое ОЗУ представляет собой массив ячеек, каждая из которых содержит транзистор и крошечный конденсатор. Конденсаторы могут быть заряженными и разряженными, что позволяет хранить нули и единицы. Поскольку электрический заряд имеет тенденцию исчезать, каждый бит в динамическом ОЗУ должен **обновляться** (перезаряжаться) каждые несколько миллисекунд, чтобы предотвратить утечку данных. Поскольку об обновлении должна заботиться внешняя логика, динамическое ОЗУ требует более сложного сопряжения, чем статическое, хотя этот недостаток компенсируется большим объемом.

Поскольку динамическому ОЗУ нужен только 1 транзистор и 1 конденсатор на бит (статическому ОЗУ требуется в лучшем случае 6 транзисторов на бит), динамическое ОЗУ имеет очень высокую плотность записи (много битов на одну микросхему). По этой причине основная память почти всегда строится на основе динамических ОЗУ. Однако динамические ОЗУ работают очень медленно (время доступа занимает десятки наносекунд). Таким образом, сочетание кэш-памяти на основе статического ОЗУ и основной памяти на основе динамического ОЗУ соединяет в себе преимущества обоих устройств.

Существует несколько типов динамических ОЗУ. Самый древний тип, который все еще используется, — **FPM (Fast Page Mode — быстрый постраничный**

режим). Это ОЗУ представляет собой матрицу битов. Аппаратное обеспечение представляет адрес строки, а затем — адреса столбцов (мы описывали этот процесс, когда говорили об устройстве памяти, показанном на рис. 3.30, *б*).

FPM постепенно замещается **EDO**[1] (**Extended Data Output — память с расширенными возможностями вывода**), которая позволяет обращаться к памяти еще до того, как закончилось предыдущее обращение. Такой конвейерный режим не ускоряет доступ к памяти, но зато увеличивает пропускную способность, выдавая больше слов в секунду.

И FPM, и EDO являются асинхронными. В отличие от них так называемое **синхронное динамическое ОЗУ** управляется одним синхронизирующим сигналом. Данное устройство представляет собой гибрид статического и динамического ОЗУ. Синхронное динамическое ОЗУ часто используется при производстве кэш-памяти большого объема. Возможно, данная технология в будущем станет наиболее предпочтительной и в изготовлении основной памяти.

ОЗУ — не единственный тип микросхем памяти. Во многих случаях данные должны сохраняться, даже если питание отключено (например, если речь идет об игрушках, различных приборах и машинах). Более того, после установки ни программы, ни данные не должны изменяться. Эти требования привели к появлению **ПЗУ (постоянных запоминающих устройств)**, которые не позволяют изменять и стирать хранящуюся в них информацию (ни умышленно, ни случайно). Данные записываются в ПЗУ в процессе производства. Для этого изготавливается трафарет с определенным набором битов, который накладывается на фоточувствительный материал, а затем открытые (или закрытые) части поверхности вытравливаются. Единственный способ изменить программу в ПЗУ — поменять целую микросхему.

ПЗУ стоят гораздо дешевле ОЗУ, если заказывать их большими партиями, чтобы оплатить расходы на изготовление трафарета. Однако они не допускают изменений после выпуска с производства, а между подачей заказа на ПЗУ и его выполнением может пройти несколько недель. Чтобы компаниям было проще разрабатывать новые устройства, основанные на ПЗУ, были выпущены **программируемые ПЗУ**. В отличие от обычных ПЗУ, их можно программировать в условиях эксплуатации, что позволяет сократить время выполнения заказа. Многие программируемые ПЗУ содержат массив крошечных плавких перемычек. Можно пережечь определенную перемычку, если выбрать нужную строку и нужный столбец, а затем приложить высокое напряжение к определенному выводу микросхемы.

Следующая разработка этой линии — **стираемое программируемое ПЗУ**, которое можно не только программировать в условиях эксплуатации, но и стирать с него информацию. Если кварцевое окно в данном ПЗУ подвергать воздействию сильного ультрафиолетового света в течение 15 минут, все биты установятся на 1. Если нужно сделать много изменений во время одного этапа проектирования, стираемые ПЗУ гораздо экономичнее, чем обычные программируемые ПЗУ, поскольку их можно использовать многократно. Стираемые программируемые ПЗУ обычно устроены так же, как статические ОЗУ. Например, микросхема 27C040 имеет структуру, которая показана на рис. 3.30, *а*, а такая структура типична для статического ОЗУ.

---

[1] Динамическая память типа EDO вытеснила обычную динамическую память, работающую в режиме FPM, в середине 90-х годов. — *Примеч. научн. ред.*

Следующий этап — **электронно-перепрограммируемое ПЗУ**, с которого можно стирать информацию, прилагая к нему импульсы, и которое не нужно для этого помещать в специальную камеру, чтобы подвергнуть воздействию ультрафиолетовых лучей. Кроме того, чтобы перепрограммировать данное устройство, его не нужно вставлять в специальный аппарат для программирования, в отличие от стираемого программируемого ПЗУ. Но с другой стороны, самые большие электронно-перепрограммируемые ПЗУ в 64 раза меньше обычных стираемых ПЗУ, и работают они в два раза медленнее. Электронно-перепрограммируемые ПЗУ не могут конкурировать с динамическими и статическими ОЗУ, поскольку они работают в 10 раз медленнее, их емкость в 100 раз меньше и они стоят гораздо дороже. Они используются только в тех ситуациях, когда необходимо сохранение информации при выключении питания.

Более современный тип электронно-перепрограммируемого ПЗУ — **флэш-память**. В отличие от стираемого ПЗУ, которое стирается под воздействием ультрафиолетовых лучей, и от электронно-программируемого ПЗУ, которое стирается по байтам, флэш-память стирается и записывается блоками. Как и любое электронно-перепрограммируемое ПЗУ, флэш-память можно стирать, не вынимая ее из микросхемы. Многие изготовители производят небольшие печатные платы, содержащие десятки мегабайтов флэш-памяти. Они используются для хранения изображений в цифровых камерах и для других целей. Возможно, когда-нибудь флэш-память вытеснит диски, что будет грандиозным шагом вперед, учитывая время доступа в 100 нс. Основной технической проблемой в данный момент является то, что флэш-память изнашивается после 10 000 стираний, а диски могут служить годами независимо от того, сколько раз они перезаписывались. Краткое описание различных типов памяти дано в табл. 3.2.

Таблица 3.2. Характеристики различных видов памяти

Тип запоминающего устройства	Категория	Стирание записи	Изменение информации по байтам	Энергозависимость	Применение
Статическое ОЗУ (SRAM)	Чтение/запись	Электрическое	Да	Да	Кэш-память второго уровня
Динамическое ОЗУ (DRAM)	Чтение/запись	Электрическое	Да	Да	Основная память
ПЗУ (ROM)	Только чтение	Невозможно	Нет	Нет	Устройства большого размера
Программируемое ПЗУ (PROM)	Только чтение	Невозможно	Нет	Нет	Устройства небольшого размера
Стираемое программируемое ПЗУ (EPROM)	Преимущественно чтение	Ультрафиолетовый свет	Нет	Нет	Моделирование устройств
Электронно-перепрограммируемое ПЗУ (EEPROM)	Преимущественно чтение	Электрическое	Да	Нет	Моделирование устройств
Флэш-память (Flash)	Чтение/запись	Электрическое	Нет	Нет	Цифровые камеры

Микросхемы процессоров и шины

Поскольку нам уже известна некоторая информация о МИС, СИС и микросхемах памяти, то мы можем сложить все составные части вместе и изучать целые системы. В этом разделе сначала мы рассмотрим процессоры на цифровом логическом уровне, включая **цоколевку** (то есть значение сигналов на различных выводах). Поскольку центральные процессоры тесно связаны с шинами, которые они используют, мы также кратко изложим основные принципы разработки шин. В следующих разделах мы подробно опишем примеры центральных процессоров и шин для них.

Микросхемы процессоров

Все современные процессоры помещаются на одной микросхеме. Это делает вполне определенным их взаимодействие с остальными частями системы. Каждая микросхема процессора содержит набор выводов, через которые происходит обмен информацией с внешним миром. Одни выводы передают сигналы от центрального процессора, другие принимают сигналы от других компонентов, третьи делают и то и другое. Изучив функции всех выводов, мы сможем узнать, как процессор взаимодействует с памятью и устройствами ввода-вывода на цифровом логическом уровне.

Выводы микросхемы центрального процессора можно подразделить на три типа: адресные, информационные и управляющие. Эти выводы связаны с соответствующими выводами на микросхемах памяти и микросхемах устройств ввода-вывода через набор параллельных проводов (так называемую шину). Чтобы вызвать команду, центральный процессор сначала посылает в память адрес этой команды по адресным выводам. Затем он запускает одну или несколько линий управления, чтобы сообщить памяти, что ему нужно, например, прочитать слово. Память выдает ответ, помещая требуемое слово на информационные выводы процессора и посылая сигнал о том, что это сделано. Когда центральный процессор получает данный сигнал, он принимает слово и выполняет вызванную команду.

Команда может требовать чтения или записи слов, содержащих данные. В этом случае весь процесс повторяется для каждого дополнительного слова. Как происходит процесс чтения и записи, мы подробно рассмотрим ниже. Важно понимать, что центральный процессор обменивается информацией с памятью и устройствами ввода-вывода, подавая сигналы на выводы и принимая сигналы на входы. Другого способа обмена информацией не существует.

Число адресных выводов и число информационных выводов — два ключевых параметра, которые определяют производительность процессора. Микросхема, содержащая m адресных выводов, может обращаться к 2^m ячейкам памяти. Обычно m равно 16, 20, 32 или 64. Микросхема, содержащая n информационных выводов, может считывать или записывать n-битное слово за одну операцию. Обычно n равно 8, 16, 32, 36 или 64. Центральному процессору с 8 информационными выводами понадобится 4 операции, чтобы считать 32-битное слово, тогда как процессор, имеющий 32 информационных вывода, может сделать ту же работу в одну

операцию. Следовательно, микросхема с 32 информационными выводами работает гораздо быстрее, но и стоит гораздо дороже.

Кроме адресных и информационных выводов каждый процессор содержит выводы управления. Выводы управления регулируют и синхронизируют поток данных к процессору и от него, а также выполняют другие разнообразные функции. Все процессоры содержат выводы для питания (обычно +3,3 В или +5 В), «земли» и синхронизирующего сигнала (меандра). Остальные выводы разнятся от процессора к процессору. Тем не менее выводы управления можно разделить на несколько основных категорий:

1. Управление шиной.
2. Прерывание.
3. Арбитраж шины.
4. Состояние.
5. Разное.

Ниже мы кратко опишем каждую из этих категорий. Когда мы будем рассматривать микросхемы Pentium II, UltraSPARC II и picoJava II, мы дадим более подробную информацию. Схема типичного центрального процессора, в котором используются эти типы сигналов, изображена на рис. 3.31.

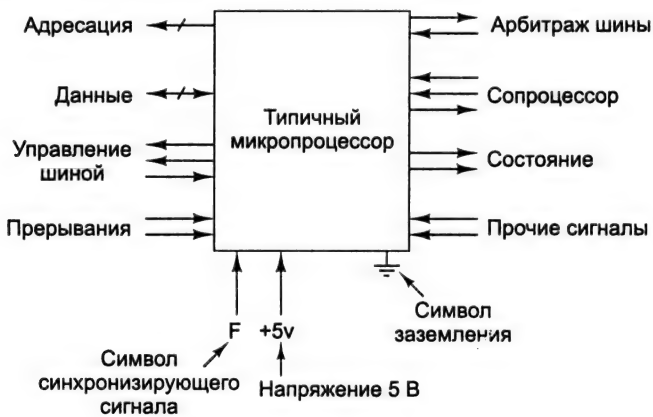


Рис. 3.31. Цоколевка типичного центрального процессора. Стрелочки указывают входные и выходные сигналы. Короткие диагональные линии указывают на наличие нескольких выводов. Для конкретных процессоров будет дано число этих выводов

Выводы управления шиной по большей части представляют собой выходы из центрального процессора в шину (и следовательно, входы в микросхемы памяти и микросхемы устройств ввода-вывода). Они сообщают, что процессор хочет считать информацию из памяти, или записать информацию в память, или сделать что-нибудь еще.

Выводы прерывания — это входы из устройств ввода-вывода в процессор. В большинстве систем процессор может дать сигнал устройству ввода-вывода начать операцию, а затем приступить к какому-нибудь другому действию, пока устройство

ввода-вывода выполняет свою работу. Когда устройство ввода-вывода заканчивает свою работу, контроллер ввода-вывода посылает сигнал на один из выводов прерывания, чтобы прервать работу процессора и заставить его обслуживать устройство ввода-вывода (например, проверять ошибки ввода-вывода). Некоторые процессоры содержат выходной вывод, чтобы подтверждать получение сигнала прерывания.

Выводы разрешения конфликтов в шине нужны для того, чтобы регулировать поток информации в шине, то есть не допускать таких ситуаций, когда два устройства пытаются воспользоваться шиной одновременно. В целях разрешения конфликтов центральный процессор считается устройством.

Некоторые центральные процессоры могут работать с различными сопроцессорами (например, с графическими процессорами, процессорами с плавающей точкой и т. п.). Чтобы обеспечить обмен информации между процессором и сопроцессором, нужны специальные выводы для передачи сигналов.

Кроме этих выводов у некоторых процессоров есть различные дополнительные выводы. Одни из них выдают или принимают информацию о состоянии, другие нужны для перезагрузки компьютера, а третьи — для обеспечения совместимости со старыми микросхемами устройств ввода-вывода.

Шины

Шина — это группа проводников, соединяющих различные устройства. Шины можно разделить на группы в соответствии с выполняемыми функциями. Они могут быть внутренними по отношению к процессору и служить для передачи данных в АЛУ и из АЛУ, а могут быть внешними по отношению к процессору и связывать процессор с памятью или устройствами ввода-вывода. Каждый тип шины обладает определенными свойствами, и к каждому из них предъявляются определенные требования. В этом и следующих разделах мы сосредоточимся на шинах, которые связывают центральный процессор с памятью и устройствами ввода-вывода. В следующей главе мы подробно рассмотрим внутренние шины процессора.

Первые персональные компьютеры имели одну внешнюю шину, которая называлась **системной шиной**. Она состояла из нескольких медных проводов (от 50 до 100), которые встраивались в материнскую плату. На материнской плате находились разъемы на одинаковых расстояниях друг от друга для микросхем памяти и устройств ввода-вывода. Современные персональные компьютеры обычно содержат специальную шину между центральным процессором и памятью и по крайней мере еще одну шину для устройств ввода-вывода. На рис. 3.32 изображена система с одной шиной памяти и одной шиной ввода-вывода.

В литературе шины обычно изображаются в виде жирных стрелок, как показано на этом рисунке. Разница между жирной и нежирной стрелкой небольшая. Когда все биты одного типа, например адресные или информационные, рисуется обычная стрелка. Когда включаются адресные линии, линии данных и управления, используется жирная стрелка.

Хотя разработчики процессоров могут использовать любой тип шины для микросхемы, должны быть введены четкие правила о том, как работает шина, и все

устройства, связанные с шиной, должны подчиняться этим правилам, чтобы платы, которые выпускаются третьими лицами, подходили к системной шине. Эти правила называются **протоколом шины**. Кроме того, должны существовать определенные технические требования, чтобы платы от третьих производителей подходили к каркасу для печатных плат и имели разъемы, соответствующие материнской плате механически и с точки зрения мощностей, синхронизации и т. д.

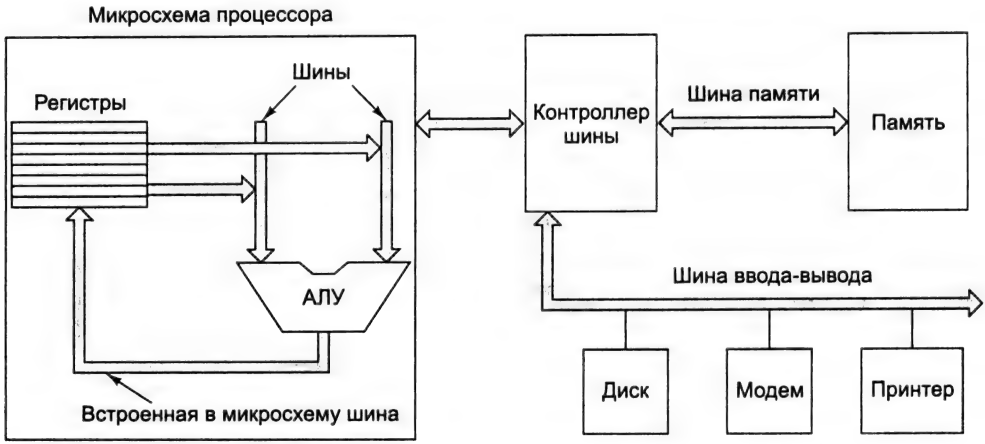


Рис. 3.32. Компьютерная система с несколькими шинами

Существует ряд широко используемых в компьютерном мире шин. Приведем несколько примеров: Omnibus (PDP-8), Unibus (PDP-11), IBM PC (PC/XT), ISA (PC/AT), EISA (80386), Microchannel (PC/2), PCI (различные персональные компьютеры), SCSI (различные персональные компьютеры и рабочие станции), Nubus (Macintosh), Universal Serial Bus (современные персональные компьютеры), FireWire (бытовая электроника), VME (оборудование в кабинетах физики) и Camac (физика высоких энергий). Может быть, все стало бы намного проще, если бы все шины, кроме одной, исчезли с поверхности Земли (или кроме двух). К сожалению, стандартизация в этой области кажется маловероятной, и уже вложено слишком много средств во все эти несовместимые системы.

Давайте начнем с того, как работают шины. Некоторые устройства, связанные с шиной, являются активными и могут инициировать передачу информации по шине, тогда как другие являются пассивными и ждут запросов. Активное устройство называется **задающим устройством**, пассивное — **подчиненным устройством**. Когда центральный процессор требует от контроллера диска считать или записать блок информации, центральный процессор действует как задающее устройство, а контроллер диска — как подчиненное устройство. Контроллер диска может действовать как задающее устройство, когда он командует памяти принять слова, которые считал с диска. Несколько типичных комбинаций задающего и подчиненного устройств указаны в табл. 3.3. Память ни при каких обстоятельствах не может быть задающим устройством.

Таблица 3.3. Примеры задающих и подчиненных устройств

Задающее устройство	Подчиненное устройство	Пример
Центральный процессор	Память	Вызов команд и данных
Центральный процессор	Устройство ввода-вывода	Инициализация передачи данных
Центральный процессор	Сопроцессор	Передача команды от процессора к сопроцессору
Устройство ввода-вывода	Память	ПДП (прямой доступ к памяти)
Сопроцессор	Центральный процессор	Вызов сопроцессором операндов из центрального процессора

Двоичные сигналы, которые выдают устройства компьютера, часто недостаточно интенсивны, чтобы активизировать шину, особенно если она достаточно длинная и если к ней подсоединено много устройств. По этой причине большинство задающих устройств шины обычно связаны с ней через микросхему, которая называется **драйвером шины**, по существу являющуюся двоичным усилителем. Сходным образом большинство подчиненных устройств связаны с шиной **приемником шины**. Для устройств, которые могут быть и задающим, и подчиненным устройством, используется **приемопередатчик шины**. Эти микросхемы взаимодействия с шиной часто являются устройствами с тремя состояниями, что дает им возможность отсоединяться, когда они не нужны. Иногда они подключаются через **открытый коллектор**, что дает сходный эффект. Когда одно или несколько устройств на открытом коллекторе требуют доступа к шине в одно и то же время, результатом является булева операция ИЛИ над всеми этими сигналами. Такое соглашение называется **монтажным ИЛИ**. В большинстве шин одни линии являются устройствами с тремя состояниями, а другие, которым требуется свойство монтажного ИЛИ, — открытым коллектором.

Как и процессор, шина имеет адресные линии, информационные линии и линии управления. Тем не менее между выводами процессора и сигналами шины может и не быть взаимно однозначного соответствия. Например, некоторые процессоры содержат три вывода, которые выдают сигнал чтения из памяти или записи в память, или чтения устройства ввода-вывода, или записи на устройство ввода-вывода, или какой-либо другой операции. Обычная шина может содержать одну линию для чтения из памяти, вторую линию для записи в память, третью — для чтения устройства ввода-вывода, четвертую — для записи на устройство ввода-вывода и т. д. Микросхема-декодер должна тогда связывать данный процессор с такой шиной, чтобы преобразовывать 3-битный кодированный сигнал в отдельные сигналы, которые могут управлять линиями шины.

Разработка шин и принципы действия шин — это достаточно сложные вопросы, и по этому поводу написан ряд книг [128, 135, 136]. Принципиальными вопросами в разработке являются ширина шины, синхронизация шины, арбитраж шины и функционирование шины. Все эти параметры существенно влияют на скорость и пропускную способность шины. В следующих четырех разделах мы рассмотрим каждый из них.

Ширина шины

Ширина шины — самый очевидный параметр при разработке. Чем больше адресных линий содержит шина, тем к большему объему памяти может обращаться процессор. Если шина содержит n адресных линий, тогда процессор может использовать ее для обращения к 2^n различным ячейкам памяти. Для памяти большой емкости необходимо много адресных линий. Это звучит достаточно просто.

Проблема заключается в том, что для широких шин требуется больше проводов, чем для узких. Они занимают больше физического пространства (например, на материнской плате), и для них нужны разъемы большего размера. Все эти факторы делают шину дорогостоящей. Следовательно, необходим компромисс между максимальным размером памяти и стоимостью системы. Система с шиной, содержащей 64 адресные линии, и памятью в 2^{32} байт будет стоить дороже, чем система с шиной, содержащей 32 адресные линии, и такой же памятью в 2^{32} байт. Дальнейшее расширение не бесплатное.

Многие разработчики систем недальновидны, что приводит к неприятным последствиям. Первая модель IBM PC содержала процессор 8088 и 20-битную адресную шину (рис. 3.33, *а*). Шина позволяла обращаться к 1 Мбайт памяти.

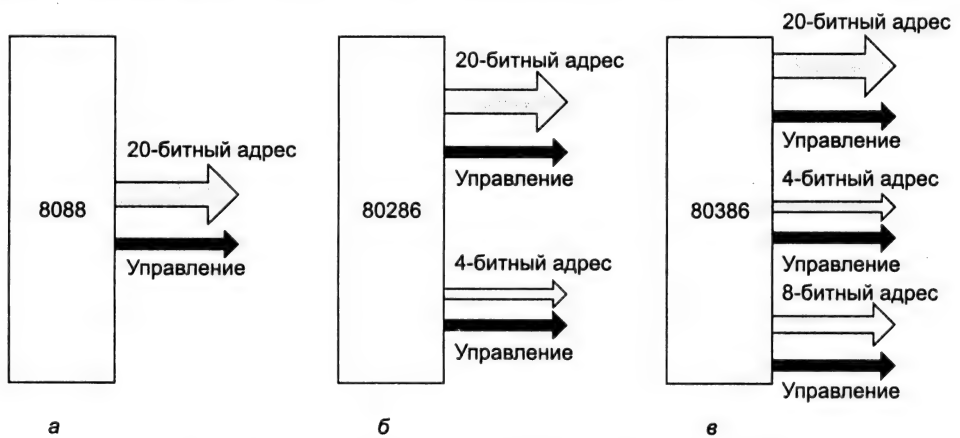


Рис. 3.33. Расширение адресной шины с течением времени

Когда появился следующий процессор (80286), Intel решил увеличить адресное пространство до 16 Мбайт, поэтому пришлось добавить еще 4 линии (не нарушая изначальные 20 по причинам совместимости с более старыми версиями), как показано на рис. 3.33, *б*. К сожалению, пришлось также добавить линии управления для новых адресных линий. Когда появился процессор 80386, было добавлено еще 8 адресных линий и, естественно, несколько линий управления, как показано на рис. 3.33, *в*. В результате получилась шина EISA. Однако было бы лучше, если бы с самого начала имелось 32 линии.

С течением времени увеличивается не только число адресных линий, но и число информационных линий. Хотя это происходит по несколько другой причине. Можно увеличить пропускную способность шины двумя способами: сократить время цикла шины (сделать большее количество передач в секунду) или увели-

чить ширину шины данных (то есть увеличить количество битов за одну передачу). Можно повысить скорость работы шины, но сделать это довольно сложно, поскольку сигналы на разных линиях передаются с разной скоростью (это явление называется **перекосом шины**). Чем быстрее работает шина, тем больше перекос.

При увеличении скорости работы шины возникает еще одна проблема: в этом случае она не будет совместимой с более старыми версиями. Старые платы, разработанные для более медленной шины, не могут работать с новой. Такая ситуация невыгодна для владельцев и производителей старых плат. Поэтому обычно для увеличения производительности просто добавляются новые линии, как показано на рис. 3.33. Как вы понимаете, в этом тоже есть свои недостатки. IBM PC и его последователи, например, начали с 8 информационных линий, затем перешли к 16, а затем к 32, и все это в одной и той же шине.

Чтобы обойти эту проблему, разработчики иногда отдают предпочтение **мультиплексной шине**. В этой шине нет разделения на адресные и информационные линии. В ней может быть, например, 32 линии и для адресов, и для данных. Сначала эти линии используются для адресов. Затем они используются для данных. Чтобы записать информацию в память, нужно сначала передавать в память адрес, а затем данные. В случае с отдельными линиями адреса и данные могут передаваться вместе. Объединение линий сокращает ширину и стоимость шины, но система работает при этом медленнее. Поэтому разработчикам приходится взвешивать все за и против, прежде чем сделать выбор.

Синхронизация шины

Шины можно разделить на две категории в зависимости от их синхронизации. **Синхронная шина** содержит линию, которая запускается кварцевым генератором. Сигнал на этой линии представляет собой меандр с частотой обычно от 5 до 100 МГц. Любое действие шины занимает целое число так называемых **циклов шины**. **Асинхронная шина** не содержит задающего генератора. Циклы шины могут быть любой требуемой длины и необязательно одинаковы по отношению ко всем парам устройств. Ниже мы рассмотрим каждый тип шины отдельно.

Синхронные шины

В качестве примера того, как работает асинхронная шина, рассмотрим временную диаграмму на рис. 3.34. В этом примере мы будем использовать задающий генератор на 40 МГц, который дает цикл шины в 25 нс. Хотя может показаться, что шина работает медленно по сравнению с процессорами на 500 МГц и выше, не многие современные шины работают быстрее. Например, шина ISA (она встроена во все персональные компьютеры с процессором Intel) работает с частотой 8,33 МГц, и даже популярная шина PCI — с частотой 33 МГц или 66 МГц. Причины такой низкой скорости современных шин были даны выше: такие технические проблемы, как перекос шины и требование совместимости.

В нашем примере мы предполагаем, что считывание информации из памяти занимает 40 нс с того момента, как адрес стал постоянным. Как мы скоро увидим, понадобится три цикла шины, чтобы считать одно слово. Первый цикл начинается

на нарастающем фронте отрезка T_1, а третий заканчивается на нарастающем фронте отрезка T_3, как показано на рис. 3.34. Отметим, что ни один из нарастающих и задних фронтов не нарисован вертикально, потому что ни один электрический сигнал не может изменять свое значение за нулевое время. В нашем примере мы предполагаем, что для изменения сигнала требуется 1 нс. Генератор и линии ADDRESS, DATA, $\overline{\text{MREQ}}$, $\overline{\text{RD}}$, $\overline{\text{WAIT}}$ показаны в том же масштабе времени.

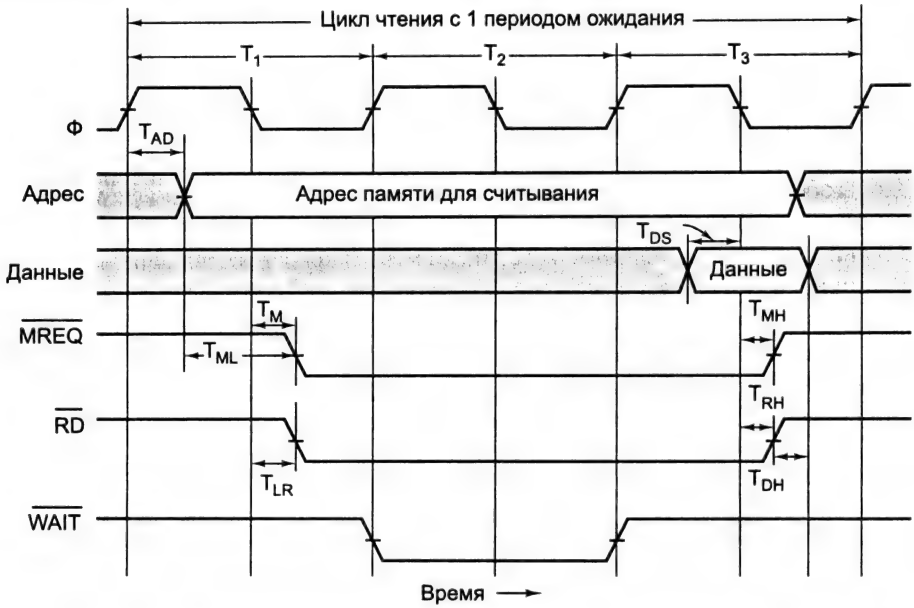


Рис. 3.34. Временная диаграмма процесса считывания на синхронной шине

Начало T_1 определяется нарастающим фронтом генератора. За часть времени T_1 центральный процессор помещает адрес нужного слово на адресные линии. Поскольку адрес представляет собой не одно значение (в отличие от генератора), мы не можем показать его в виде одной линии на схеме. Вместо этого мы показали его в виде двух линий с пересечениями там, где этот адрес меняется. Серый цвет на схеме показывает, что в этот момент не важно, какое значение принял сигнал. Используя то же соглашение, мы видим, что содержание линий данных не имеет значения до отрезка T_3.

После того как у адресных линий появляется возможность приобрести новое значение, устанавливаются сигналы $\overline{\text{MREQ}}$ и $\overline{\text{RD}}$. Первый указывает, что осуществляется доступ к памяти, а не к устройству ввода-вывода, а второй — что осуществляется чтение, а не запись. Поскольку считывание информации из памяти занимает 40 нс после того, как адрес стал постоянным (часть первого цикла), память не может передать требуемые данные за период T_2. Чтобы центральный процессор не ожидал поступления данных, память устанавливает линию $\overline{\text{WAIT}}$ в начале отрезка T_2. Это действие вводит **периоды ожидания** (дополнительные циклы шины), до тех пор пока память не сбросит сигнал $\overline{\text{WAIT}}$. В нашем примере вводится один период ожидания (T_2), поскольку память работает слишком медленно. В начале

T_3, когда есть уверенность в том, что память получит данные в течение текущего цикла, сигнал $\overline{WAIT}$ сбрасывается.

Во время первой половины T_3 память помещает данные на информационные линии. На заднем фронте T_3 центральный процессор стробирует (то есть считывает) информационные линии, сохраняя их значения во внутреннем регистре. Считав данные, центральный процессор сбрасывает сигналы $\overline{MREQ}$ и $\overline{RD}$. В случае необходимости на следующем нарастающем фронте может начаться еще один цикл памяти.

Далее проясняется значение восьми символов на временной диаграмме (см. рис. 3.34 и табл. 3.4). T_{AD}, например, — это временной интервал между нарастающим фронтом T_1 и установкой адресных линий. В соответствии с требованиями синхронизации $T_{AD} \leq 11$ нс. Значит, производитель процессора гарантирует, что во время любого цикла считывания центральный процессор будет выдавать требуемый адрес в пределах 11 нс от середины нарастающего фронта T_1.

Таблица 3.4. Некоторые временные характеристики процесса считывания на синхронной шине

Символ	Значение	Минимум, нс	Максимум, нс
T_{AD}	Задержка выдачи адреса		11
T_{ML}	Промежуток между стабилизацией адреса и установкой сигнала $\overline{MREQ}$	6	
T_M	Промежуток между задним фронтом синхронизирующего сигнала в цикле T_1 и установкой сигнала $\overline{MREQ}$		8
T_{RL}	Промежуток между задним фронтом синхронизирующего сигнала в цикле T_1 и установкой сигнала $\overline{RD}$		8
T_{DS}	Период передачи данных до заднего фронта синхронизирующего сигнала	5	
T_{MH}	Промежуток между задним фронтом синхронизирующего сигнала в цикле T_3 и сбросом сигнала $\overline{MREQ}$		8
T_{RH}	Промежуток между задним фронтом синхронизирующего сигнала в цикле T_3 и сбросом сигнала $\overline{RD}$		8
T_{DH}	Период продолжения передачи данных с момента сброса сигнала $\overline{RD}$	0	

Условия синхронизации также требуют, чтобы данные поступали на информационные линии по крайней мере за 5 нс (T_{DS}) до заднего фронта T_3, чтобы дать данным время установиться до того, как процессор стробирует их. Сочетание ограничений на T_{AD} и T_{DS} означает, что в худшем случае в распоряжении памяти будет только 62,5–11–5=46,5 нс с момента появления адреса и до момента, когда нужно выдавать данные. Поскольку достаточно 40 нс, память даже в самом худшем случае может всегда ответить за период T_3. Если памяти для считывания требуется 50 нс, то необходимо ввести второй период ожидания, и тогда память ответит в течение T_3.

Требования синхронизации гарантируют, что адрес будет установлен по крайней мере за 6 нс до того, как появится сигнал $\overline{\text{MREQ}}$. Это время может быть важно в том случае, если $\overline{\text{MREQ}}$ запускает выбор элемента памяти, поскольку некоторые типы памяти требуют некоторого времени на установку адреса до выбора элемента памяти. Ясно, что разработчику системы не следует выбирать микросхему памяти, на установку которой нужно 10 нс.

Ограничения на T_M и T_{RL} означают, что $\overline{\text{WREQ}}$ и $\overline{\text{RD}}$ будут установлены в пределах 8 нс от заднего фронта T_1. В худшем случае у микросхемы памяти после установки $\overline{\text{MREQ}}$ и $\overline{\text{RD}}$ останется всего 25+25–8–5=37 нс на передачу данных по шине. Это ограничение дополнительно по отношению к интервалу в 40 нс и не зависит от него.

T_{MH} и T_{RH} определяют, сколько времени требуется на отмену сигналов $\overline{\text{MREQ}}$ и $\overline{\text{RD}}$ после того, как данные стробированы. Наконец, T_{DH} определяет, сколько времени память должна держать данные на шине после снятия сигнала $\overline{\text{RD}}$. В нашем примере при данном процессоре память может удалить данные с шины, как только сбрасывается сигнал $\overline{\text{RD}}$; при других процессорах, однако, данные могут сохраняться еще некоторое время.

Необходимо подчеркнуть, что наш пример представляет собой сильно упрощенную версию реальных временных ограничений. В действительности должно определяться гораздо больше таких ограничений. Тем не менее этот пример наглядно демонстрирует, как работает синхронная шина.

Отметим, что сигналы управления могут задаваться или с помощью низкого, или с помощью высокого напряжения. Что является более удобным в каждом конкретном случае, должен решать разработчик, хотя, по существу, выбор произволен.

Асинхронные шины

Хотя достаточно удобно использовать синхронные шины благодаря дискретным временным интервалам, здесь все же есть некоторые проблемы. Например, если процессор и память способны закончить передачу за 3,1 цикла, они вынуждены продлить ее до 4,0 циклов, поскольку неполные циклы запрещены.

Еще хуже то, что если однажды был выбран определенный цикл шины и в соответствии с ним были разработаны память и карты ввода-вывода, то в будущем трудно делать технологические усовершенствования. Например, предположим, что через несколько лет после выпуска системы, изображенной на рис. 3.34, появилась новая память с временем доступа не 40, а 20 нс. Это избавило бы нас от периода ожидания и увеличило скорость работы машины. Теперь представим, что появилась память с временем доступа 10 нс. При этом улучшения производительности уже не будет, поскольку в данной разработке минимальное время для чтения — 2 цикла.

Если синхронная шина соединяет ряд устройств, одни из которых работают быстро, а другие медленно, шина подстраивается под самое медленное устройство, а более быстрые не могут использовать свой полный потенциал.

По этой причине были разработаны асинхронные шины, то есть шины без задающего генератора, как показано на рис. 3.35. Здесь ничего не привязывается к генератору. Когда задающее устройство устанавливает адрес, $\overline{\text{MREQ}}$, $\overline{\text{RD}}$ и любой

другой требуемый сигнал, он выдает специальный сигнал, который мы будем называть $\overline{\text{MSYN}}$ (Master SYNchronization). Когда подчиненное устройство получает этот сигнал, оно начинает выполнять свою работу настолько быстро, насколько это возможно. Когда работа закончена, устройство выдает сигнал $\overline{\text{SSYN}}$ (Slave SYNchronization).



Рис. 3.35. Работа асинхронной шины

Сигнал $\overline{\text{SSYN}}$ означает для задающего устройства, что данные доступны. Он фиксирует их, а затем отключает адресные линии вместе с $\overline{\text{MREQ}}$, $\overline{\text{RD}}$ и $\overline{\text{MSYN}}$. Отмена сигнала $\overline{\text{MSYN}}$ означает для подчиненного устройства, что цикл закончен, поэтому устройство отменяет сигнал $\overline{\text{SSYN}}$, и все возвращается к первоначальному состоянию, когда все сигналы отменены.

Стрелочки на временных диаграммах асинхронных шин (а иногда и синхронных шин) показывают причину и следствие какого-либо действия (рис. 3.35). Установка сигнала $\overline{\text{MSYN}}$ приводит к запуску информационных линий, а также к установке сигнала $\overline{\text{SSYN}}$. Установка сигнала $\overline{\text{SSYN}}$, в свою очередь, вызывает отключение адресных линий, $\overline{\text{MREQ}}$, $\overline{\text{RD}}$ и $\overline{\text{MSYN}}$. Наконец, отключение $\overline{\text{MSYN}}$ вызывает отключение $\overline{\text{SSYN}}$, и на этом процесс считывания заканчивается.

Набор таких взаимообусловленных сигналов называется **полным квитированием**. Здесь, в сущности, наблюдается 4 события:

1. Установка сигнала $\overline{\text{MSYN}}$.
2. Установка сигнала $\overline{\text{SSYN}}$ в ответ на сигнал $\overline{\text{MSYN}}$.
3. Отмена сигнала $\overline{\text{MSYN}}$ в ответ на сигнал $\overline{\text{SSYN}}$.
4. Отмена сигнала $\overline{\text{SSYN}}$ в ответ на отмену сигнала $\overline{\text{MSYN}}$.

Следует уяснить, что взаимообусловленность сигналов не зависит от синхронизации. Каждое событие вызывается предыдущим событием, а не импульсами генератора. Если какая-то пара двух устройств (задающего и подчиненного) работает медленно, это никак не повлияет на следующую пару устройств, которая работает гораздо быстрее.

Преимущества асинхронной шины очевидны, но в действительности большинство шин являются синхронными. Дело в том, что синхронную систему построить проще, чем асинхронную. Центральный процессор просто выдает сигналы, а память просто реагирует на них. Здесь нет никакой причинно-следственной связи, но если компоненты выбраны удачно, все будет работать и без квитирования. Кроме того, в разработку синхронных шин сделано очень много вложений.

Арбитраж шины

До этого момента мы неявно предполагали, что существует только одно задающее устройство шины — центральный процессор. В действительности микросхемы ввода-вывода могут становиться задающим устройством при считывании информации из памяти и записи информации в память. Кроме того, они могут вызывать прерывания. Сопроцессоры также могут становиться задающим устройством шины. Возникает вопрос: «Что происходит, когда задающим устройством шины могут стать два или несколько устройств одновременно?» Чтобы предотвратить хаос, который может при этом возникнуть, нужен специальный механизм — так называемый **арбитраж шины**.

Механизмы арбитража могут быть централизованными или децентрализованными. Рассмотрим сначала централизованный арбитраж. Простой пример централизованного арбитража показан на рис. 3.36, *а*. В данном примере один арбитр шины определяет, чья очередь следующая. Часто бывает, что арбитр встроен в микросхему процессора, но иногда требуется отдельная микросхема. Шина содержит одну линию запроса (монтажное ИЛИ), которая может запускаться одним или несколькими устройствами в любое время. Арбитр не может определить, сколько устройств запрашивают шину. Он может определять только наличие или отсутствие запросов.

Когда арбитр видит запрос шины, он запускает линию предоставления шины. Эта линия последовательно связывает все устройства ввода-вывода (как в елочной гирлянде). Когда физически ближайшее к арбитру устройство воспринимает сигнал предоставления шины, оно проверяет, нет ли запроса шины. Если запрос есть, устройство пользуется шиной, но не распространяет сигнал предоставления дальше по линии. Если запроса нет, устройство передает сигнал предоставления шины следующему устройству. Это устройство тоже проверяет, есть ли запрос, и действует соответствующим образом в зависимости от наличия или отсутствия запроса. Передача сигнала предоставления шины продолжается до тех пор, пока какое-нибудь устройство не воспользуется предоставленной шиной. Такая система называется **системой последовательного опроса**. При этом приоритеты устройств зависят от того, насколько близко они находятся к арбитру. Ближайшее к арбитру устройство обладает главным приоритетом.

Чтобы обойти такую систему, в которой приоритеты зависят от расстояния от арбитра, в некоторых шинах устраивается несколько уровней приоритета. На каждом уровне приоритета есть линия запроса шины и линия предоставления шины. На рис. 3.36, *б* изображено 2 уровня (хотя в действительности шины обычно содержат 4, 8 или 16 уровней). Каждое устройство связано с одним из уровней

запроса шины, причем, чем выше уровень приоритета, тем больше устройств привязано к этому уровню. На рис. 3.36, *б* можно видеть, что устройства 1, 2 и 4 используют приоритет 1, а устройства 3 и 5 — приоритет 2.

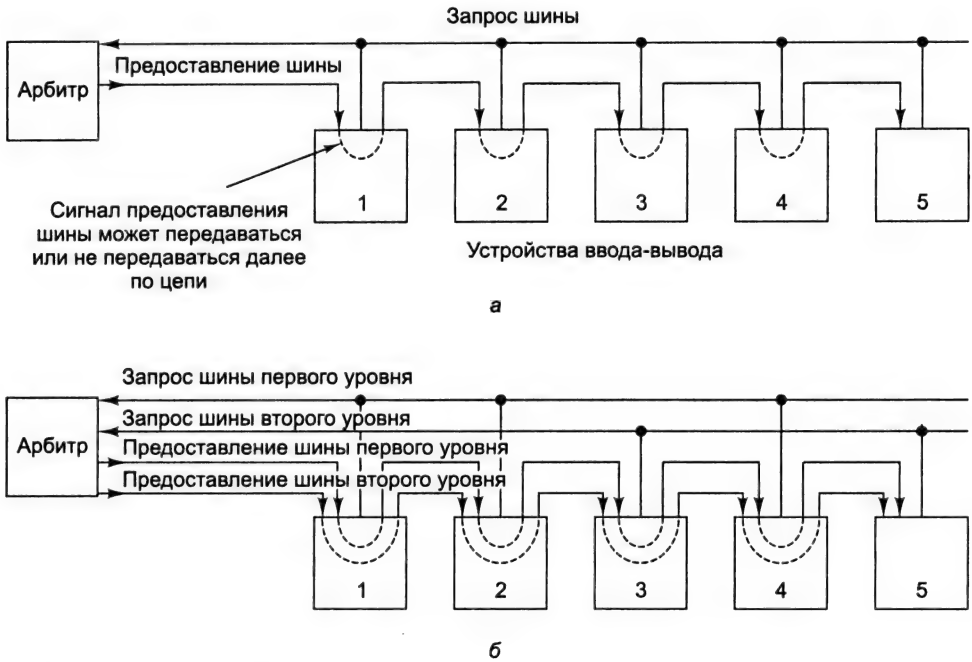


Рис. 3.36. Одноуровневый централизованный арбитраж шины с использованием системы последовательного опроса (*а*); двухуровневый централизованный арбитраж (*б*)

Если одновременно запрашивается несколько уровней приоритета, арбитр предоставляет шину самому высокому уровню. Среди устройств одинакового приоритета используется система последовательного опроса. На рис. 3.36, *б* видно, что в случае конфликта устройство 2 «побеждает» устройство 4, а устройство 4 «побеждает» устройство 3. Устройство 5 имеет низший приоритет, поскольку оно находится в самом конце самого нижнего уровня.

Линия предоставления шины второго уровня необязательно должна последовательно связывать устройства 1 и 2, поскольку они не могут посылать на нее запросы. Однако гораздо проще провести все линии предоставления шины через все устройства, чем соединять устройства особым образом в зависимости от их приоритетов.

Некоторые арбитры содержат третью линию, которая запускается, как только устройство принимает сигнал предоставления шины, и берет шину в свое распоряжение. Как только запускается эта линия подтверждения приема, линии запроса и предоставления шины могут быть отключены. В результате другие устройства могут запрашивать шину, пока первое устройство использует ее. К тому моменту, когда закончится текущая передача, следующее задающее устройство уже будет выбрано. Это устройство может начать работу, как только отключается линия

подтверждения приема. С этого момента начинается следующий арбитраж. Такая структура требует наличия дополнительной линии и большего количества логических схем в каждом устройстве, но зато при этом циклы шины используются рациональнее.

В системах, где память связана с главной шиной, центральный процессор должен завершать работу со всеми устройствами ввода-вывода практически на каждом цикле шины. Чтобы решить эту проблему, можно предоставить центральному процессору самый низкий приоритет. При этом шина будет предоставляться процессору только в том случае, если она не нужна ни одному другому устройству. Центральный процессор всегда может подождать, а устройства ввода-вывода должны получить доступ к шине как можно быстрее, чтобы не потерять данные. Диски, вращающиеся с высокой скоростью, тоже не могут ждать. Во многих современных компьютерах память помещается на одну шину, а устройства ввода-вывода — на другую, поэтому им не приходится завершать работу, чтобы предоставить доступ к шине.

Возможен также децентрализованный арбитраж шины. Например, компьютер может содержать 16 приоритетных линий запроса шины. Когда устройству нужна шина, оно запускает свою линию запроса. Все устройства контролируют все линии запроса, поэтому в конце каждого цикла шины каждое устройство может определить, обладает ли оно в данный момент высшим приоритетом и, следовательно, разрешено ли линии пользоваться шиной в следующем цикле. Такой метод требует наличия большего количества линий, но зато не требует затрат на арбитра. Он также ограничивает число устройств числом линий запроса.

При другом типе децентрализованного арбитража используется только три линии независимо от того, сколько устройств имеется в наличии (рис. 3.37). Первая линия — монтажное ИЛИ. Она используется для запроса шины. Вторая линия называется BUSY. Она запускается текущим задающим устройством шины. Третья линия используется для арбитража шины. Она последовательно соединяет все устройства. Начало цепи связано с источником питания с напряжением 5 В.

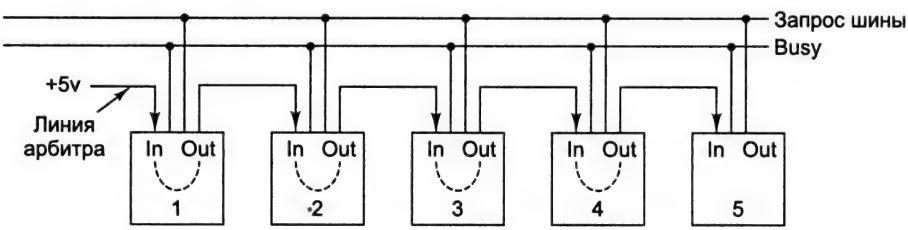


Рис. 3.37. Децентрализованный арбитраж шины

Когда шина не требуется ни одному из устройств, линия арбитра передает сигнал всем устройствам. Чтобы получить доступ к шине, устройство сначала проверяет, свободна ли шина, и установлен ли сигнал арбитра IN. Если сигнал IN не установлен, устройство не может стать задающим устройством шины. В этом случае оно сбрасывает сигнал OUT. Если сигнал IN установлен, устройство также сбрасывает сигнал OUT, в результате чего следующее устройство не получает сигнал IN и, в свою очередь, сбрасывает сигнал OUT. Следовательно, все следующие по цепи устройства не получают сигнал IN и сбрасывают сигнал OUT. В результа-

те остается только одно устройство, у которого сигнал IN установлен, а сигнал OUT сброшен. Оно становится задающим устройством шины, запускает линию BUSY и сигнал OUT и начинает передачу данных.

Немного поразмыслив, можно обнаружить, что из всех устройств, которым нужна шина, доступ к шине получает самое левое. Такая система сходна с системой последовательного опроса, только в данном случае нет арбитра, поэтому она стоит дешевле и работает быстрее. К тому же не возникает проблем со сбоями арбитра.

Принципы работы шины

До этого момента мы обсуждали только обычные циклы шины, когда задающее устройство (обычно центральный процессор) считывает информацию из подчиненного устройства (обычно из памяти) или записывает в него информацию. Однако существует еще несколько типов циклов шины. Давайте рассмотрим некоторые из них.

Обычно за раз передается одно слово. При использовании кэш-памяти желательно сразу вызывать всю строку кэш-памяти (то есть 16 последовательных 32-битных слов). Часто передача блоками может быть более эффективна, чем такая последовательная передача информации. Когда начинается чтение блока, задающее устройство сообщает подчиненному устройству, сколько слов нужно передать (например, помещая общее число слов на информационные линии в период T_1). Вместо того чтобы выдать в ответ одно слово, задающее устройство выдает одно слово в течение каждого цикла до тех пор, пока не будет передано требуемое количество слов. На рис. 3.38 изображена такая же схема, как и на рис. 3.34, только здесь появился дополнительный сигнал $\overline{\text{BLOCK}}$, который указывает, что запрашивается передача блока. В данном примере считывание блока из 4 слов занимает 6 циклов вместо 12.

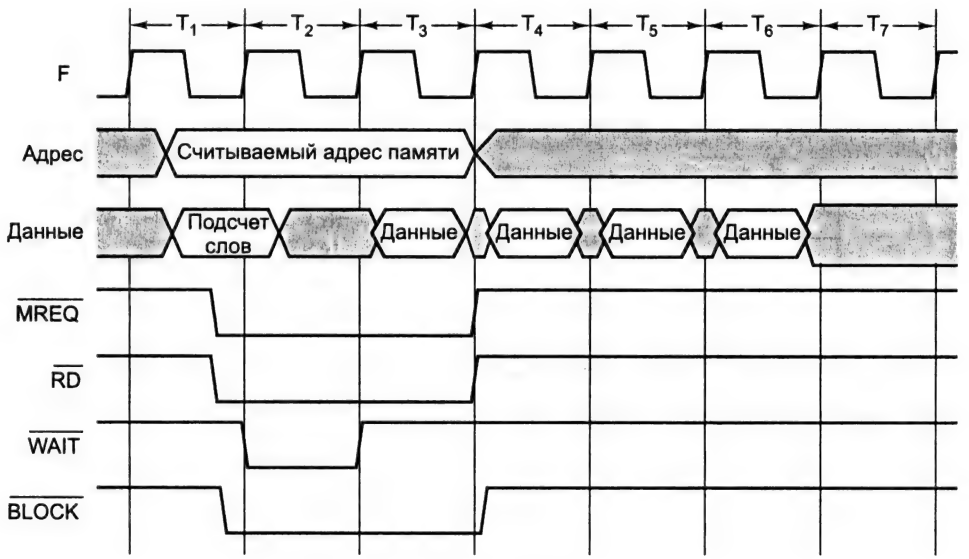


Рис. 3.38. Передача блока данных

Существуют также другие типы циклов шины. Например, если речь идет о системах с двумя или несколькими центральными процессорами на одной шине, нужно быть уверенным, что в конкретный момент только один центральный процессор может использовать определенную структуру данных в памяти. Чтобы упорядочить этот процесс, в памяти должна содержаться переменная, которая принимает значение 0, когда центральный процессор использует структуру данных, и 1, когда структура данных не используется. Если центральному процессору нужно получить доступ к структуре данных, он должен считать переменную, и если она равна 0, придать ей значение 1. Проблема заключается в том, что два центральных процессора могут считать переменную на последовательных циклах шины. Если каждый процессор видит, что переменная равна 0, а затем каждый процессор меняет значение переменной на 1, как будто только он один использует эту структуру данных, то такая последовательность событий ведет к хаосу.

Чтобы предотвратить такую ситуацию, в многопроцессорных системах предусмотрен специальный цикл шины, который дает возможность любому процессору считать слово из памяти, проверить и изменить его, а затем записать обратно в память; весь этот процесс происходит без освобождения шины. Такой цикл не дает возможности другим центральным процессорам использовать шину и, следовательно, мешать работе первого процессора.

Еще один важный цикл шины — цикл для осуществления прерываний. Когда центральный процессор командует устройству ввода-вывода произвести какое-то действие, он ожидает прерывания после завершения работы. Для сигнала прерывания нужна шина.

Поскольку может сложиться ситуация, когда несколько устройств одновременно хотят произвести прерывание, здесь имеют место те же проблемы разрешения конфликтных ситуаций, что и в обычных циклах шины. Чтобы избежать таких проблем, нужно каждому устройству приписать определенный приоритет и использовать централизованный арбитр для распределения приоритетов. Существует стандартный контроллер прерываний, который широко используется. В компьютерах IBM PC и последующих моделях применяется микросхема Intel 8259A. Она изображена на рис. 3.39.

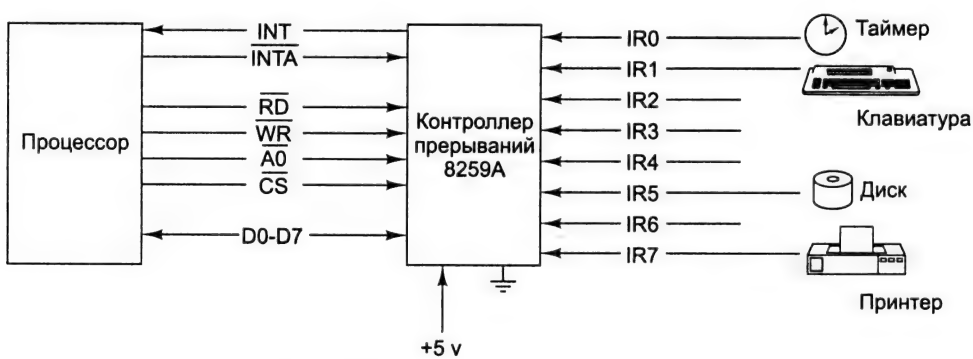


Рис. 3.39. Контроллер прерывания 8259A

До восьми контроллеров ввода-вывода могут быть непосредственно связаны с восемью входами IRx (Interrupt Request — запрос прерывания) микросхемы 8259A. Когда любое из этих устройств хочет произвести прерывание, оно запускает свою линию входа. Если активизируется один или несколько входов, контроллер 8259A выдает сигнал INT (INTerrupt — прерывание), который подается на соответствующий вход центрального процессора. Когда центральный процессор способен произвести прерывание, он посылает микросхеме 8259A импульс через вывод INTA (INTerrupt Acknowledge — подтверждение прерывания). В этот момент микросхема 8259A должна определить, на какой именно вход поступил сигнал прерывания. Для этого она помещает номер входа на информационную шину. Эта операция требует наличия особого цикла шины. Центральный процессор использует этот номер для обращения в таблицу указателей, которую называют таблицей **векторов прерывания**, чтобы найти адрес процедуры, производящей соответствующее прерывание.

Микросхема 8259A содержит несколько регистров, которые центральный процессор может считывать и записывать, используя обычные циклы шины и выводы $\overline{\text{RD}}$ (ReaD — чтение), $\overline{\text{WR}}$ (WRite — запись), $\overline{\text{CS}}$ (Chip Select — выбор элемента памяти) и $\overline{\text{A0}}$. Когда программное обеспечение обработало прерывание и готово получить следующее, оно записывает специальный код в один из регистров, который вызывает сброс сигнала INT микросхемой 8259A, если не появляется другая задержка прерывания. Регистры также могут записываться для того, чтобы ввести микросхему 8259A в один из нескольких режимов, и для выполнения некоторых других функций.

Когда присутствует более восьми устройств ввода-вывода, микросхемы 8259A могут быть соединены каскадно. В самой экстремальной ситуации все восемь входов могут быть связаны с выходами еще восьми микросхем 8259A, соединяя до 64 устройств ввода-вывода в двухступенчатую систему прерывания. Микросхема 8259A содержит несколько выводов для управления каскадированием, но мы их опустили ради простоты.

Хотя мы никоим образом не исчерпали все вопросы разработки шин, материал, изложенный выше, дает достаточно информации для общего понимания принципов работы шины и того, как центральный процессор взаимодействует с шиной. А теперь мы перейдем от общего к частному и рассмотрим несколько конкретных примеров процессоров и их шин.

Примеры центральных процессоров

В этом разделе мы рассмотрим процессоры Pentium II, Ultra SPARC II и picoJava на уровне аппаратного обеспечения.

Pentium II

Pentium II — прямой потомок процессора 8088, который использовался в первой модели IBM PC. Хотя Pentium II очень сильно отличается от процессора 8088 (первый содержит 7,5 млн транзисторов, а второй — всего 29 000), он полностью

совместим с 8088 и может выполнять программы, написанные для 8088 (не говоря уже о программах для всех процессоров, появившихся между Pentium II и 8088).

С точки зрения программного обеспечения, Pentium II представляет собой 32-разрядную машину. Он содержит ту же архитектуру системы команд, что и процессоры 80386, 80486, Pentium и Pentium Pro, включая те же регистры, те же команды и такую же встроенную систему с плавающей точкой стандарта IEEE 754.

С точки зрения аппаратного обеспечения, Pentium II представляет собой нечто большее, поскольку он может обращаться к 64 Гбайт физической памяти и передавать данные в память и из памяти блоками по 64 бита. Программист не видит этих передач по 64 бита, но такая машина работает быстрее, чем 32-разрядная.

На микроархитектурном уровне Pentium II представляет собой Pentium Pro с командами MMX. Команды вызываются из памяти заранее и разбиваются на микрооперации. Эти микрооперации хранятся в буфере, и как только одна из них получает необходимые ресурсы для выполнения, она может начаться. Если в одном цикле может начинаться несколько микроопераций, Pentium II является суперскалярной машиной.

Pentium II имеет двухуровневую кэш-память. Кэш-память первого уровня содержит 16 Кбайт для команд и 16 Кбайт для данных, а смежная кэш-память второго уровня — еще 512 Кбайт. Строка кэш-памяти состоит из 32 байт. Тактовая частота кэш-памяти второго уровня в два раза меньше тактовой частоты центрального процессора. Тактовая частота центрального процессора — 233 МГц и выше.

В системах с процессором Pentium II используются две внешние шины, обе они синхронные. Шина памяти используется для доступа к главному динамическому ОЗУ; шина PCI используется для сообщения с устройствами ввода-вывода. Иногда к шине PCI подсоединяется **унаследованная** (то есть прежняя) шина, чтобы можно было подключать старые периферические устройства.

Система Pentium II может содержать один или два центральных процессора, которые разделяют общую память. В системе с двумя процессорами может возникнуть одна неприятная ситуация. Слово, считанное в одну из микросхем кэш-памяти и измененное там, может не записаться обратно в память, и если второй процессор попытается считать это слово, он получит неправильное значение. Чтобы предотвратить такую ситуацию, существуют специальные системы поддержки.

Pentium II существенно отличается от своих предшественников компоновкой. Все процессоры, начиная с 8088 и заканчивая Pentium Pro, были обычными микросхемами с выводами по бокам или снизу, которые вставлялись в разъемы. Микропроцессор Pentium II представляет собой так называемый **SEC (Single Edge Cartridge — картридж с однорядным расположением контактов)**. Как видно из рис. 3.40, этот картридж представляет собой довольно большую пластиковую коробку, содержащую центральный процессор, двухуровневую кэш-память и торцевой соединитель для передачи сигналов. Он содержит 242 контакта.

Хотя у Intel были все основания перейти к такой модели, она повлекла за собой проблему, которую компания не могла предвидеть. Многие покупатели имеют привычку разбирать компьютер и искать микросхему процессора. Однако в первых системах Pentium II покупатели не могли найти процессор и громко жаловались («У моего компьютера нет процессора!»). Компания Intel разрешила эту проблему, приклеив изображение процессора (голограмму) на следующие модели SEC.

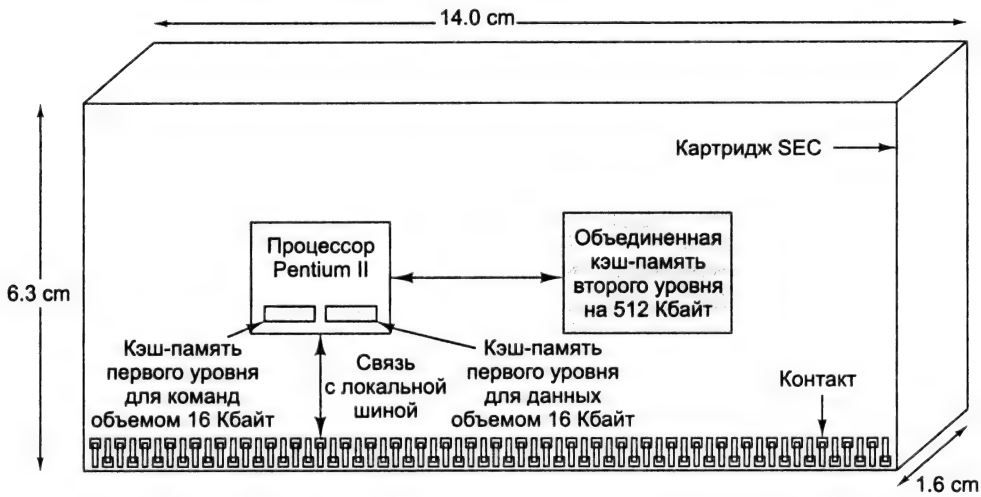


Рис. 3.40. Компоновка SEC

Главная проблема, связанная с процессором Pentium II, — управление режимом электропитания. Количество выделяемого тепла зависит от тактовой частоты, а выделяемая мощность при этом разнится от 30 до 50 Вт. Это огромная цифра для компьютерной микросхемы. Чтобы получить представление, что такое 50 Вт, поднесите руку к электрической лампочке в 50 Вт, которая включена уже некоторое время (только не дотрагивайтесь до нее). По этой причине SEC снабжен радиатором, чтобы рассеивать накопившееся тепло. Когда Pentium II перестанет быть рабочим процессором, его можно будет использовать в качестве нагревателя.

В соответствии с законами физики все, что выделяет большое количества тепла, должно потреблять большое количество энергии. В случае с портативным компьютером, который работает от батареи с ограниченным зарядом, потребление большого количества энергии нежелательно. Чтобы решить эту проблему, компания Intel нашла способ вводить центральный процессор в режим пониженного энергоснабжения (состояние «сна»), если он не выполняет никаких действий, и вообще отключать его (вводить в состояние «глубокого сна»), если есть вероятность, что он не будет выполнять никаких действий некоторое время. В последнем случае значения кэш-памяти и регистров сохраняются, а тактовый генератор и все внутренние блоки отключаются. Видит ли Pentium II сны во время «глубокого сна», науке пока не известно.

Цоколевка процессора Pentium II

Из 242 контактов картриджа SEC 170 используются для сигналов, 27 для питания (с различной мощностью), 35 для «земли» и еще 10 остаются на будущее. Для некоторых логических сигналов используется два и более выводов (например, для запроса адреса памяти), поэтому существует только 53 типа выводов. Цоколевка в несколько упрощенном виде представлена на рис. 3.41. С левой стороны рисунка показано 6 основных групп сигналов шины памяти; с правой стороны расположены прочие сигналы. Заглавными буквами обозначены названия самих сигналов,

а строчными — общие названия для групп связанных сигналов (в последнем случае только первая буква заглавная).

Компания Intel использует одно соглашение, которое важно понимать. Поскольку микросхемы разрабатываются с использованием компьютеров, нужно каким-то образом представлять названия сигналов в виде текста ASCII. Использовать черту над названиями сигналов, запускаемых низким напряжением, слишком сложно, вместо этого компания Intel помещает после названия сигнала знак #. Например, вместо обозначения $\overline{\text{BPRI}}$ используется BPRI#. Как видно из рисунка, большинство сигналов Pentium II запускаются низким напряжением.

Давайте рассмотрим различные типы сигналов. Начнем с сигналов шины. Первая группа сигналов используется для запроса шины (то есть для арбитража). Сигнал BPRI# позволяет устройству с высоким приоритетом получить доступ к шине раньше других устройств. Сигнал LOCK# позволяет центральному процессору не предоставлять остальным устройствам доступа к шине, пока работа не будет закончена.

Центральный процессор или другое задающее устройство шины может производить запрос на доступ к шине, используя следующую группу сигналов. Адреса состоят из 36 бит, но три последних бита должны всегда быть равны 0, и следовательно, они не имеют собственных выводов, поэтому A# содержит только 33 вывода. Все передачи состоят из 8 байтов. Поскольку адрес содержит 36 бит, максимальный объем памяти составляет 2^{36}, то есть 64 Гбайт.

Когда адрес передается на шину, устанавливается сигнал ADS#. Этот сигнал сообщает целевому объекту (например, памяти), что задействованы адресные линии. На линиях REQ# запускается цикл шины определенного типа (например, считывание слова или запись блока). Два сигнала четности нужны для проверки A#, а третий — для проверки ADS# и REQ#. Подчиненное устройство использует пять специальных линий для сообщения об ошибках четности. Эти же линии используются всеми устройствами для сообщения о каких-либо других ошибках.

Группа сигналов для отслеживания адресов, по которым происходило изменение данных, используется в многопроцессорных системах. Эти сигналы позволяют процессору обнаружить, есть ли слово, которое ему нужно, в кэш-памяти другого процессора. Как процессор Pentium II отслеживает эти адреса, мы рассмотрим в главе 8.

Ответные сигналы передаются от подчиненного к задающему устройству. Сигнал RS# содержит код состояния. Сигнал TRDY# показывает, что подчиненное устройство (целевой объект) готово принять данные от задающего устройства. Эти сигналы также проверяются на четность.

Последняя группа сигналов нужна для передачи данных. Сигнал D# используется, чтобы поместить 8 байтов данных на шину. Когда они туда помещаются, выдается сигнал DRDY# (сигнал наличия данных в шине). Он сообщает устройствам, что шина в данный момент занята.

Сигнал RESET# нужен для перезагрузки процессора в случае сбоя. Pentium II может осуществлять прерывания тем же способом, что и процессор 8088 (это требуется в целях совместимости), или использовать новую систему прерывания с устройством **APIC (Advanced Programmable Interrupt Controller — встроенный контроллер прерываний)**.

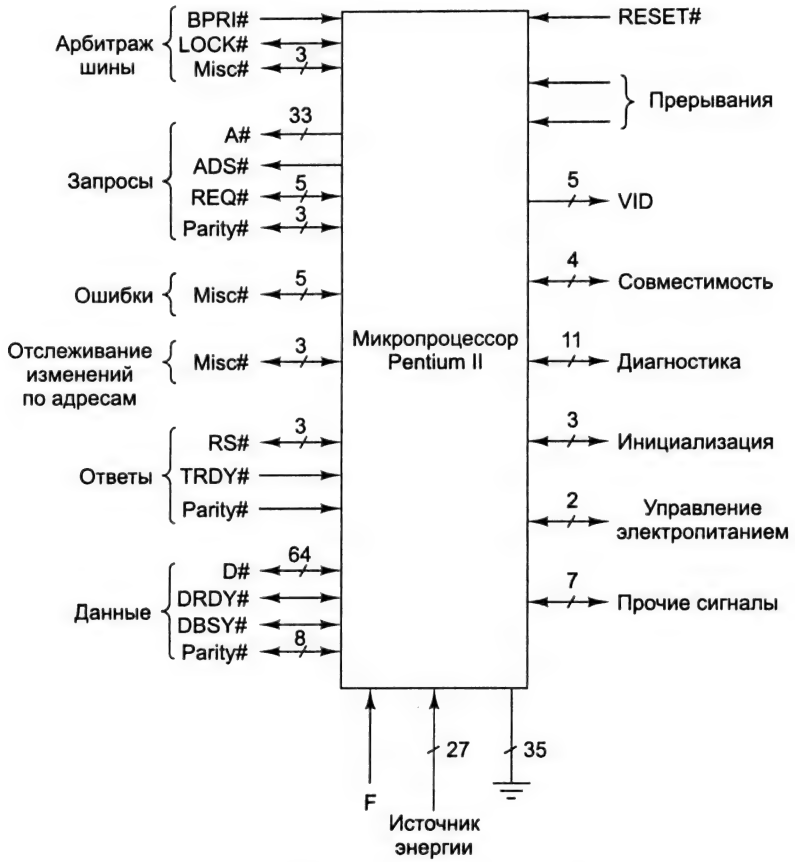


Рис. 3.41. Цоколевка процессора Pentium II. Официальные названия отдельных сигналов приведены заглавными буквами. Строчными буквами даны общие названия групп связанных сигналов или описания сигналов

Pentium II может работать при разном напряжении. Сигналы VID используются для автоматического выбора напряжения источника питания. Сигналы совместимости нужны для работы с устройствами более старых моделей, которые изначально предназначены для процессора 8088. Группа сигналов диагностики содержит сигналы для проверки и отладки системы в соответствии со стандартом IEEE 1149.1 JTAG. Сигналы инициализации связаны с загрузкой системы. Сигналы управления режимом электропитания позволяют процессору входить в состояние «сна» или «глубокого сна». Наконец, оставшаяся группа содержит сигналы разного рода. Сюда относится, например, сигнал, который выдается центральным процессором, если его внутренняя температура превышает 130°C (266°F). Хотя если температура центрального процессора превышает 130°C, он уже, вероятно, мечтает о выходе на пенсию и хочет служить в качестве нагревателя.

Конвейерный режим шины памяти процессора Pentium II

Современные процессоры, например Pentium II, работают гораздо быстрее современных динамических ОЗУ. Чтобы процессор не простаивал, необходима макси-

мально возможная производительность памяти. По этой причине шина памяти процессора Pentium II работает в конвейерном режиме, при этом в шине происходит одновременно 8 операций. Понятие конвейера мы рассматривали в главе 2, когда говорили о конвейерных процессорах. Отметим, что память тоже может быть конвейерной.

Обращения процессора к памяти, которые называются **транзакциями**, имеют 6 стадий:

1. Фаза арбитража шины.
2. Фаза запроса.
3. Фаза сообщения об ошибке.
4. Фаза проверки на наличие нужного слова в другом процессоре.
5. Фаза ответа.
6. Фаза передачи данных.

Наличие всех шести фаз необязательно. На фазе арбитража шины определяется, какое из задающих устройств будет следующим. На фазе запроса на шину передается адрес. На фазе сообщения об ошибке подчиненное устройство передает сигнал об ошибке четности в адресе или о наличии каких-либо других неполадок. На следующей фазе центральный процессор проверяет, нет ли нужного ему слова в другом процессоре. Эта стадия нужна только в многопроцессорных системах. В следующей фазе задающее устройство узнает, где взять необходимые данные. На последней стадии осуществляется передача данных.

В системе с процессором Pentium II на каждой стадии используются определенные сигналы, отличные от сигналов других стадий, поэтому каждая из них не зависит от остальных. Шесть групп необходимых сигналов показаны в левой части рис. 3.41. Например, один из процессоров может пытаться получить доступ к шине, используя сигналы арбитража. Как только процессор получает право на доступ к шине, он освобождает эти линии шины и занимает линии запроса. Тем временем другой процессор или какое-нибудь устройство ввода-вывода может войти в фазу арбитража шины и т. д. На рис. 3.42 показано, как осуществляется одновременно несколько транзакций.

Фаза арбитража шины на рис. 3.42 не показана, поскольку она не всегда нужна. Например, если устройство, обладающее в данный момент шиной (часто это центральный процессор), хочет произвести еще одну транзакцию, ему не требуется заново получать доступ к шине. Ему нужно запрашивать шину заново только в том случае, если он уступает ее другому устройству. Транзакции 1 и 2 обычные: пять фаз за пять циклов шины. Во время транзакции 3 вводится более длительная фаза передачи данных (поскольку, например, требуется передать целый блок или нужно ввести режим ожидания). Вследствие этого транзакция 4 не может начать фазу передачи данных сразу после стадии ответа. Стадия передачи данных начинается только после того, как исчезнет сигнал DBSY#. Фаза ответа в транзакции 5 также может занимать несколько циклов шины, что задерживает транзакцию 6. Наконец, мы видим, что в транзакции 7 также происходит задержка, поскольку она уже появилась ранее. В действительности же маловероятно, что центральный процессор будет пытаться начать новую транзакцию на каждом цикле шины, поэтому простои не такие уж длительные.

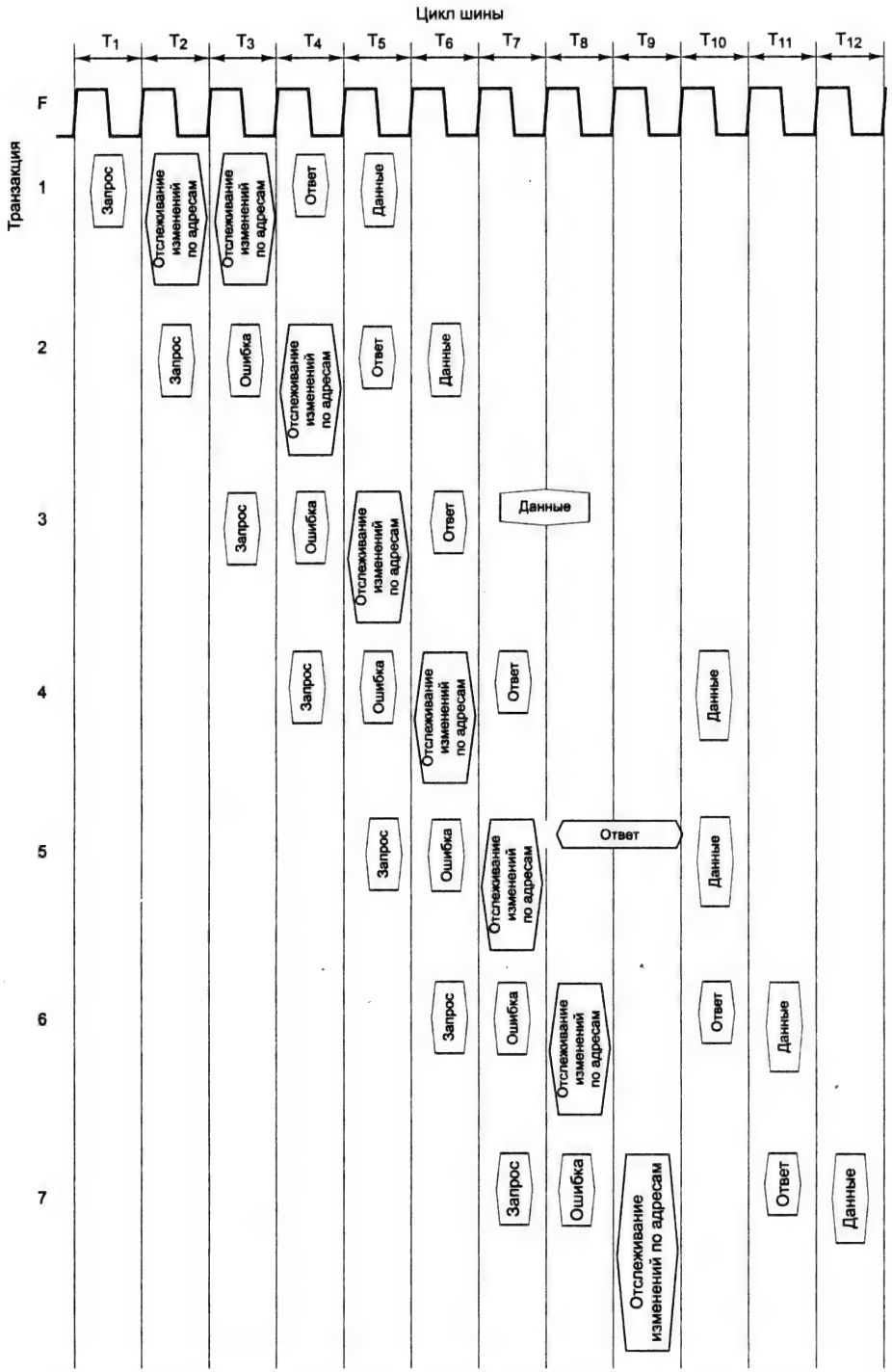


Рис. 3.42. Конвейерный режим шины памяти в системе с процессором Pentium II

UltraSPARC II

В качестве второго примера процессора возьмем семейство UltraSPARC (производитель — компания Sun). Семейство UltraSPARC — это серия 64-разрядных процессоров SPARC. Эти процессоры полностью соответствуют архитектуре Version 9 SPARC, которая также подходит для 64-разрядных процессоров. Они используются в рабочих станциях и серверах Sun, а также во многих других системах. Семейство включает в себя процессоры UltraSPARC I, UltraSPARC II и UltraSPARC III, которые имеют сходную архитектуру, но различаются датой выпуска и тактовой частотой. Ниже мы будем говорить о процессоре UltraSPARC II, поскольку нам нужен конкретный пример, но изложенная информация по большей части имеет силу и для других типов UltraSPARC.

UltraSPARC II представляет собой машину типа RISC. Он полностью совместим с 32-разрядным SPARC V8. Единственное, чем UltraSPARC II отличается от SPARC V9, — это наличием команд VIS, которые разработаны для графических приложений, кодировки MPEG в реальном времени и т. п.

Процессор UltraSPARC II был разработан для создания 4-узловых мультипроцессоров с разделенной памятью без добавления внешних схем, а также для создания более крупных мультипроцессоров с минимальным добавлением внешних схем. Иными словами, в каждую микросхему UltraSPARC II включены связующие элементы, необходимые для построения мультипроцессора.

В отличие от структуры Pentium II SEC, процессор UltraSPARC II представляет собой относительно большую самостоятельную микросхему, содержащую 5,4 млн транзисторов. Микросхема содержит 787 выводов, расположенных снизу, как показано на рис. 3.43. Такое большое число выводов объясняется, с одной стороны, использованием 64 битов для адресов и 128 битов для данных. С другой стороны, это объясняется особенностями работы кэш-памяти. Кроме того, многие выводы являются резервными. Число 787 было выбрано для того, чтобы промышленность могла производить стандартные модули. Компании, вероятно, считают простое число выводов счастливым.

Процессор UltraSPARC II содержит 2 внутренних блока кэш-памяти: 16 Кбайт для команд и 16 Кбайт для данных. Как и у Pentium II, здесь вне кристалла процессора расположена кэш-память второго уровня, но, в отличие от Pentium II, процессор UltraSPARC II не упакован в один картридж с кэш-памятью второго уровня, поэтому разработчики вправе выбирать любые микросхемы для кэш-памяти второго уровня.

Решение объединить кэш-память второго уровня с процессором или разделить ее с процессором обусловлено выбором между различными техническими преимуществами, а также особенностями компаний Intel и Sun. Внешняя кэш-память более гибкая (кэш-память процессора UltraSPARC II можно расширить с 512 Кбайт до 16 Мбайт; кэш-память процессора Pentium II имеет фиксированный объем 512 Кбайт), но при этом она работает медленнее из-за того, что расположена дальше от процессора. Для обращения к внешней кэш-памяти требуется больше сигналов (у картриджа SEC нет контактов для связи с кэш-памятью, поскольку в данном случае кэш-память встроена прямо в картридж), но среди 787 выводов процессора UltraSPARC II обязательно должны быть выводы для управления кэш-памятью.

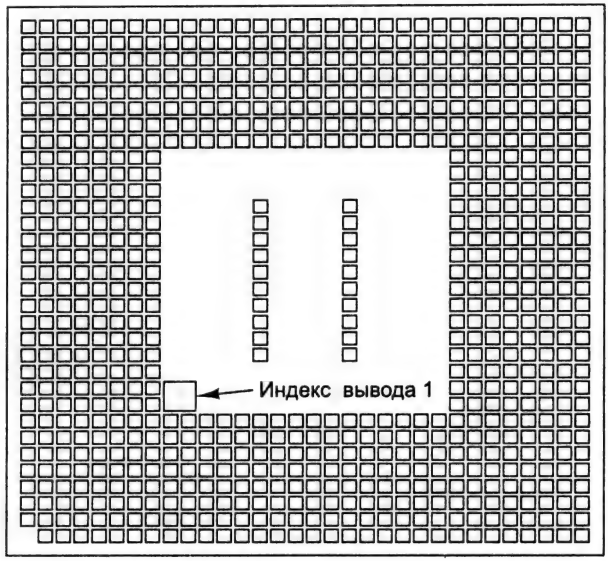


Рис. 3.43. Микросхема процессора UltraSPARC II

Что касается производственных особенностей, компания Intel является поставщиком полупроводниковых приборов, поэтому у нее есть возможность разрабатывать и выпускать собственные микросхемы кэш-памяти второго уровня и связывать их с центральным процессором через собственный интерфейс с высокими техническими характеристиками. Компания Sun, напротив, является производителем компьютеров, а не микросхем. Она иногда разрабатывает собственные микросхемы (например, UltraSPARC II), но поручает их производство предприятиям, выпускающим полупроводниковые приборы (например, Texas Instruments и Fujitsu). Иногда компания Sun предпочитает использовать микросхемы, имеющиеся в продаже. Статические ОЗУ для кэш-памяти второго уровня можно приобрести у различных производителей, поэтому у компании Sun не было особой необходимости разрабатывать собственные ОЗУ. А если ОЗУ не разрабатывается специально, то нужно устанавливать кэш-память второго уровня отдельно от центрального процессора.

Большинство рабочих станций Sun содержат синхронную шину на 25 МГц, которая называется **Sbus**. К этой шине могут подсоединяться устройства ввода-вывода. Однако шина Sbus работает слишком медленно и не подходит для памяти, поэтому компания Sun придумала другой механизм для соединения процессоров UltraSPARC II с памятью: **UPA (Ultra Port Architecture — высокоскоростной пакетный коммутатор)**. UPA может воплощаться в виде шины, переключателя или сочетания того и другого. В различных рабочих станциях и серверах используются различные реализации UPA. Реализация UPA никак не зависит от процессора, поскольку интерфейс с UPA точно определен и процессор должен поддерживать (и поддерживает) именно этот интерфейс.

На рис. 3.44 мы видим ядро системы UltraSPARC II: центральный процессор, интерфейс UPA и кэш-память второго уровня (2 статических ОЗУ). На рисунке

также изображена микросхема **UDB II (UltraSPARC II Data Buffer II. Data Buffer — буфер данных)**, функции которой мы обсудим ниже. Когда процессору нужно слово из памяти, сначала он обращается к кэш-памяти первого уровня. Если он находит слово, он продолжает выполнять операции с полной скоростью. Если он не находит слово в кэш-памяти первого уровня, он обращается к кэш-памяти второго уровня.

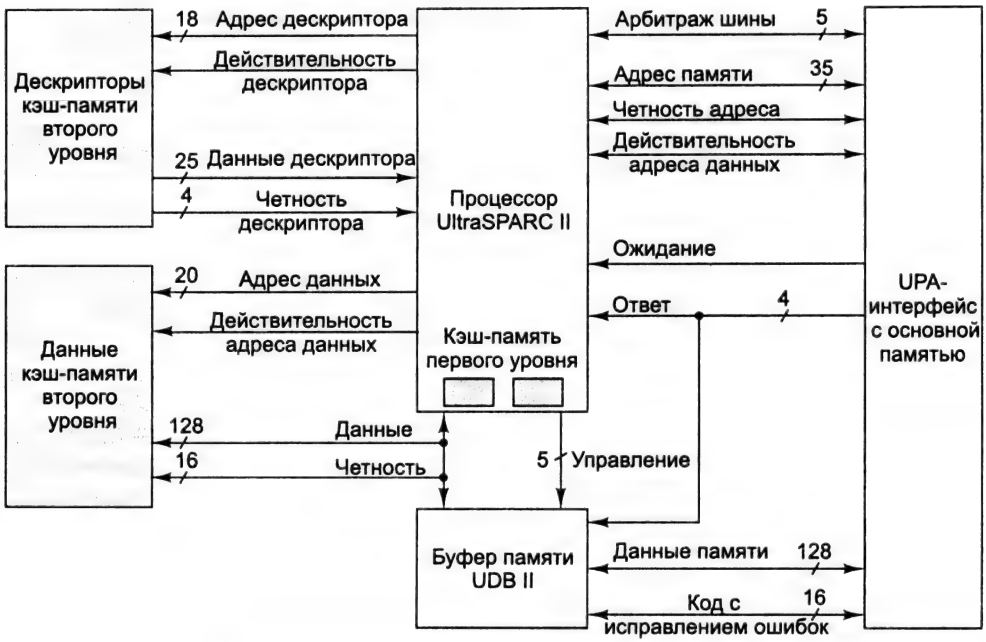


Рис. 3.44. Основная структура системы UltraSPARC II

Хотя мы в главе 4 будем подробно обсуждать работу кэш-памяти, все-таки стоит сказать здесь несколько слов об этом. Вся основная память подразделяется на строки кэш-памяти (блоки) по 64 байта. В кэш-памяти первого уровня находятся 256 наиболее часто используемых строк команд и 256 наиболее часто используемых строк данных. В кэш-памяти второго уровня содержатся строки, которые не поместились в кэш-память первого уровня. Кэш-память второго уровня содержит линии данных и команд вперемешку. Они хранятся в статическом ОЗУ, которое на рис. 3.44 обозначено прямоугольником с надписью «Данные кэш-памяти второго уровня». Система должна следить за тем, какие строки находятся в кэш-памяти второго уровня. Эта информация хранится во втором статическом ОЗУ, обозначенном на рис. 3.44 «Дескрипторы кэш-памяти второго уровня».

В случае отсутствия нужной строки в кэш-памяти первого уровня центральный процессор посылает идентификатор строки, которую он ищет (адрес дескриптора), в кэш-память второго уровня. Ответ (данные дескриптора) предоставляет центральному процессору информацию о том, есть ли нужная строка в кэш-памяти второго уровня. Если строка есть, центральный процессор получает ее. Передача данных осуществляется по 16 байтов, поэтому для пересылки целой строки в кэш-память первого уровня требуется 4 цикла.

Если требуемой строки нет в кэш-памяти второго уровня, ее нужно вызвать из основной памяти через интерфейс UPA. UPA в системе UltraSPARC II управляется централизованным контроллером. Туда поступают адресные сигналы и сигналы управления от центрального процессора (или процессоров, если их больше чем один). Чтобы получить доступ к памяти, центральный процессор должен сначала получить разрешение воспользоваться шиной. Когда шина предоставляется процессору, он получает сигнал с адресных выводов, определяет тип запроса и передает сигнал по нужному адресному выводу. (Эти выводы двунаправлены, поскольку другим процессорам в системе UltraSPARC II нужен доступ к отдаленным блокам кэш-памяти.) Адрес и тип цикла шины передаются на адресные выводы за два цикла, причем в первом цикле выдается строка, а во втором — столбец, как мы видели на рис. 3.30.

В ожидании результатов центральный процессор вполне может заниматься другой работой. Например, отсутствие нужной команды в кэш-памяти вовсе не мешает выполнению одной или нескольких команд, которые уже вызваны, и каждая из которых может обращаться к данным не из кэш-памяти. Таким образом, сразу несколько транзакций к UPA могут ожидать выполнения. Система UPA может справляться с двумя независимыми потоками транзакций (обычно это чтение и запись), каждый поток проходит с несколькими задержками. Задача централизованного контроллера — следить за всем этим и производить обращения к памяти в наиболее рациональном порядке.

Данные из памяти могут поступать блоками по 8 байтов. Они содержат 16-битный код с исправлением ошибок для большей надежности. Можно запрашивать весь блок кэш-памяти, 8 байтов или даже меньше. Все входные данные поступают в буфер UDB и хранятся там. Буфер UDB нужен для того, чтобы дать возможность центральному процессору и памяти работать асинхронно. Например, если центральному процессору необходимо записать слово или строку кэш-памяти в основную память, он может не ждать доступа к UPA, а сразу записать данные в буфер UDB, который доставит их в память позднее. UDB также генерирует код с исправлением ошибок. Отметим, что описание процессоров UltraSPARC II и Pentium II в этой книге сильно упрощено. Тем не менее мы изложили основную суть их работы.

PicoJava II

Pentium II и UltraSPARC II — процессоры с высокой производительностью, которые были разработаны для построения быстрых персональных компьютеров и рабочих станций. Существуют и другие компьютеры: так называемые встроенные системы. Именно их мы и рассмотрим кратко в этом разделе.

Не будет преувеличением сказать, что практически любое электронное устройство стоимостью более 100 долларов содержит встроенный компьютер. Телевизоры, сотовые телефоны, электронные записные книжки, микроволновые печи, видеокамеры, видеомагнитофоны, лазерные принтеры, охранные сигнализации, слуховые аппараты, электронные игры и многие другие устройства (их можно перечислять до бесконечности) управляются компьютером. При этом упор делается не на высокую производительность, а на низкую стоимость встроенного компьютера, что приводит к несколько другому соотношению преимуществ и недостатков по сравнению с процессорами, которые мы обсуждали до сих пор.

Традиционно встроенные процессоры программировались на языке ассемблер, но так как с течением времени приборы усложнялись и последствия сбоев программного обеспечения становились более серьезными, появились другие подходы. Особенно удобно использовать в качестве языка программирования для встроенных систем язык Java, поскольку он относительно прост и программы занимают мало места. К достоинствам также можно отнести независимость базовых программных средств. Однако у этого языка есть и недостатки. Во-первых, чтобы использовать язык Java во встроенных системах, требуется большой интерпретатор для выполнения кода JVM. (Программу на языке Java в код JVM преобразует специальный компилятор.) Во-вторых, процесс интерпретации занимает много времени.

Чтобы разрешить эту проблему, Sun и другие компании разработали процессоры со встроенным набором команд JVM. При таком подходе сочетаются и простота использования языка Java, и мобильность, и небольшой размер бинарного кода JVM, порождаемого компилятором, и высокая скорость выполнения операций, которая достигается благодаря особенностям аппаратного обеспечения. В этом разделе мы рассмотрим один из процессоров, который был разработан специально для встроенных систем.

Речь идет о процессоре picoJava II, который составляет основу микросхемы microJava 701. Микросхема была разработана компанией Sun, но другие компании также имеют право использовать эту разработку. Это однокристальный процессор с двумя интерфейсами шины: один из них предназначен для шины памяти шириной в 64 бита, а другой — для шины PCI, как показано на рис. 3.45. Как Pentium II и UltraSPARC II, данный процессор может содержать кэш-память первого уровня (до 16 Кбайт для команд и до 16 Кбайт для данных). Но, в отличие от этих двух процессоров, он не имеет кэш-памяти второго уровня, поскольку низкая стоимость является ключевым параметром при разработке встроенных систем. Ниже мы рассмотрим микросхему microJava II 701. Она небольшого размера: содержит всего 2 млн транзисторов плюс еще 1,5 млн для кэш-памяти.

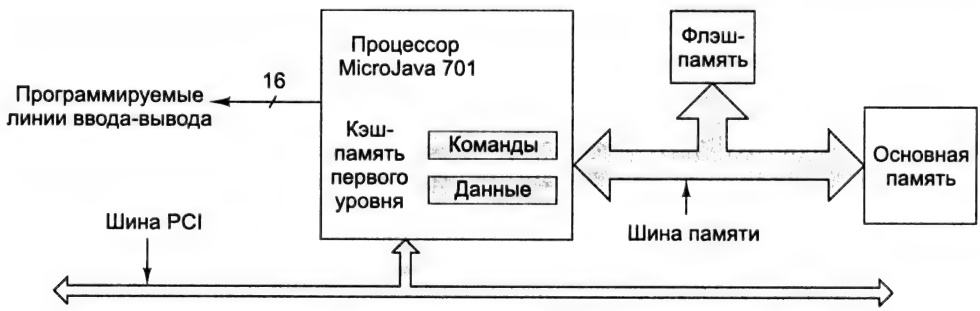


Рис. 3.45. Система microJava II 701

На рис. 3.45 видны три особенности микросхемы. Во-первых, в микросхеме microJava 701 используется шина PCI (на частоте 33 МГц или 66 МГц). Эта шина была разработана компанией Intel для использования в системах Pentium, но она подходит и для других процессоров. Преимущество шины PCI состоит в том, что она является стандартной, и поэтому не нужно каждый раз разрабатывать новую

шину. Кроме того, существует огромное количество сменных плат для этой шины. Хотя платы PCI и не играют большой роли при создании сотовых телефонов, они могут пригодиться для различных устройств большого размера (например, web-TV).

Во-вторых, система microJava II 701 обычно содержит флэш-память. Дело в том, что в прибор должна быть встроена если не вся программа, то по крайней мере ее большая часть. Флэш-память хорошо подходит для хранения программы, поэтому полезно иметь соответствующий интерфейс. Другая микросхема (на рисунке она не показана), которую можно добавить к системе, содержит последовательные и параллельные интерфейсы ввода-вывода.

В-третьих, microJava 701 содержит 16 программируемых линий ввода-вывода, которые можно связать с кнопками, переключателями и лампочками прибора. Например, у микроволновой печи обычно есть клавишная панель с цифрами и несколько дополнительных кнопок, которые можно было бы соединить непосредственно с процессором. Наличие программируемых линий ввода-вывода на процессоре исключает необходимость использования программируемых контроллеров ввода-вывода, что делает прибор проще и дешевле. В микросхему microJava 701 встроено также три программируемых тактовых генератора, которые могут быть полезны, приборы часто работают в реальном времени.

Микросхема microJava 701 выпускается в стандартном корпусе **BGA (Ball Grid Array — корпус с выводами в виде сетки крошечных шариков)**. Он содержит 316 выводов. Из них 59 выводов связаны с шиной PCI. Ниже в этой главе мы рассмотрим шину PCI подробно. Еще 123 вывода предназначены для шины памяти, среди них есть 64 двунаправленных вывода для передачи данных, а также отдельные адресные выводы. Остальные выводы используются для управления (7), синхронизирующих импульсов (3), прерываний (11), проверки (10), ввода-вывода(16). Некоторые из оставшихся выводов используются для питания и «земли», а остальные вообще не используются. Другие производители процессора picoJava II вправе выбирать иную шину, компоновку и т. д.

У данной микросхемы есть много других особенностей. Она, например, может переходить в режим ожидания (чтобы экономить заряд батарейки), она содержит встроенный контроллер прерывания, она также имеет полную поддержку для стандарта тестирования IEEE 1149.1 JTAG.

Примеры шин

Шины соединяют компьютерную систему в одно целое. В этом разделе мы рассмотрим несколько примеров шин: шину ISA, шину PCI и Universal Serial Bus (универсальную последовательную шину). Шина ISA представляет собой небольшое расширение первоначальной шины IBM PC. По соображениям совместимости она все еще используется во всех персональных компьютерах Intel[1]. Однако такие компьютеры всегда содержат еще одну шину, которая работает быстрее, чем шина ISA.

[1] Шина ISA не используется в современных компьютерах. Уже несколько лет компания Intel настоятельно рекомендует разработчикам компьютеров не использовать эту шину. — *Примеч. научн. ред.*

Это шина PCI. Она шире, чем ISA, и функционирует с более высокой тактовой частотой. Шина USB обычно используется в качестве шины ввода-вывода для периферийных устройств малого быстродействия (например, мыши и клавиатуры). В следующих разделах мы рассмотрим каждую из этих шин по очереди.

Шина ISA

Шина IBM PC была неофициальным стандартом систем с процессором 8088, поскольку практически все производители клонов скопировали ее, чтобы иметь возможность использовать в своих системах платы ввода-вывода от различных поставщиков. Шина содержала 62 сигнальные линии, из них 20 для адреса ячейки памяти, 8 для данных и по одной для сигналов считывания информации из памяти, записи информации в память, считывания с устройства ввода-вывода и записи на устройство ввода-вывода. Имелись и сигналы для запроса прерываний и их разрешения, а также для прямого доступа к памяти. Шина была очень примитивной.

Шина IBM PC встраивалась в материнскую плату персонального компьютера. На плате было несколько разъемов, расположенных на расстоянии 2 см друг от друга. В разъемы вставлялись различные платы. На платах имелись позолоченные выводы (по 31 с каждой стороны), которые физически подходили под разъемы. Через них осуществлялся электрический контакт с разъемом.

Когда компания IBM разрабатывала компьютер PC/AT с процессором 80286, она столкнулась с некоторыми трудностями. Если бы компания разработала совершенно новую 16-битную шину, многие потенциальные покупатели не стали бы приобретать этот компьютер, поскольку ни одна из сменных плат, выпускаемых другими компаниями, не подошла бы к новой машине. С другой стороны, если оставить старую шину, то новый процессор не сможет реализовать все свои возможности (например, возможность обращаться к 16 Мбайт памяти и передавать 16-битные слова).

В результате было принято решение расширить старую шину. Сменные платы персональных компьютеров содержали краевой разъем (62 контакта), но этот краевой разъем проходил не по всей длине платы. Поэтому на плате поместили еще один краевой разъем, смежный с главным. Кроме того, схемы PC/AT были разработаны таким образом, чтобы можно было подсоединять платы обоих типов. На рис. 3.46 изображена шина PC/AT.

Второй краевой разъем шины PC/AT содержит 36 линий. Из них 31 предназначена для дополнительных адресных линий, информационных линий, линий прерывания, каналов ПДП (прямого доступа к памяти), а также для питания и «земли». Остальные связаны с различиями между 8-битными и 16-битными передачами.

Когда компания IBM выпустила серию компьютеров PS/2, пришло время начать разработку шины заново. С одной стороны, это решение было обусловлено чисто техническими причинами (шина PC к тому времени уже устарела). Но с другой стороны, оно было вызвано желанием воспрепятствовать компаниям, выпускавшим клоны, которые в то время заполонили компьютерный рынок. Поэтому компьютеры PS/2 с высокой и средней производительностью были оснащены абсолютно новой шиной MCA (MicroChannel Architecture), которая была защищена патентами.

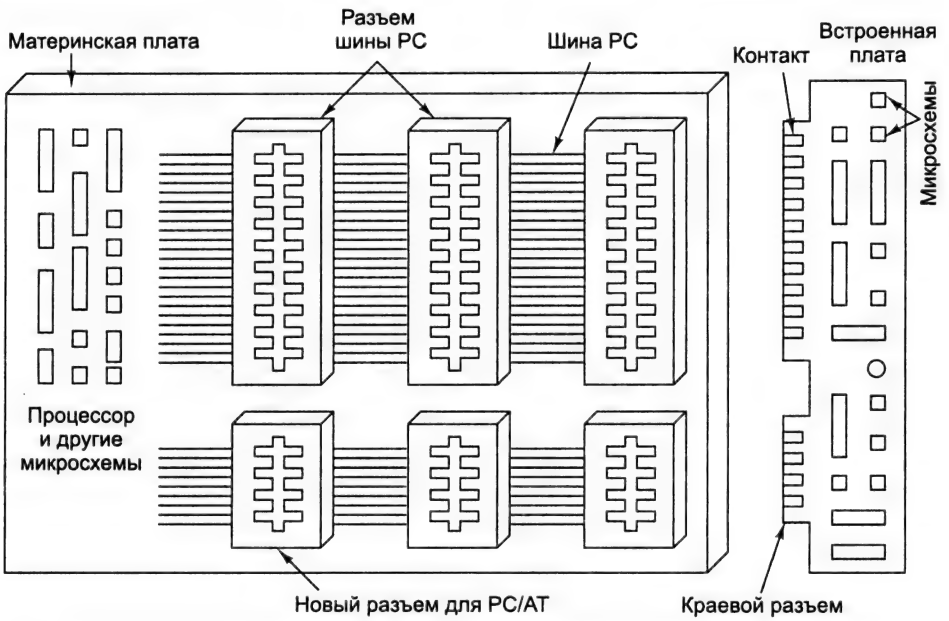


Рис. 3.46. Шина PC/AT состоит из двух компонентов: старой и новой шины

Компьютерная промышленность отреагировала на такой шаг введением своего собственного стандарта, шины **ISA (Industry Standard Architecture — стандартная промышленная архитектура)**, которая, по существу, представляет собой шину PC/AT, работающую при частоте 8,33 МГц. Преимущество такого подхода состоит в том, что при этом сохраняется совместимость с существующими машинами и платами. Отметим, что в основе этого стандарта лежит шина, разработанная компанией IBM. IBM когда-то необдуманно предоставила права на производство этой шины многим компаниям, чтобы как можно больше производителей имели возможность выпускать платы для компьютеров IBM. Однако впоследствии компании IBM пришлось об этом сильно пожалеть. Эта шина до сих пор используется во всех персональных компьютерах с процессором Intel, хотя обычно кроме нее там есть еще одна или несколько других шин. Исчерпывающее описание шины ISA можно найти в книге [127].

Позднее шина ISA была расширена до 32 разрядов. У нее появились некоторые новые особенности (например, возможность параллельной обработки). Такая шина называлась **EISA (Extended Industry Standard Architecture — расширенная архитектура промышленного стандарта)**. Для нее было разработано несколько плат.

Шина PCI

В первых компьютерах IBM PC большинство приложений имели дело с текстами. Постепенно с появлением Windows вошли в употребление графические интерфейсы пользователя. Ни одно из этих приложений не давало большой нагрузки на шину ISA. Однако с течением времени появилось множество различных приложе-

ний, в том числе игр, для которых потребовалось полноэкранное видеоизображение, и ситуация коренным образом изменилась.

Давайте произведем небольшое вычисление. Рассмотрим монитор 1024×768 для цветного движущегося изображения (3 байта/пиксел). Одно экранное изображение содержит 2,25 Мбайт данных. Для показа плавных движений требуется 30 кадров в секунду, и следовательно, скорость передачи данных должна быть 67,5 Мбайт/с. В действительности дело обстоит гораздо хуже, поскольку чтобы передать изображение, данные должны перейти с жесткого диска, компакт-диска или DVD-диска через шину в память. Затем данные должны поступить в графический адаптер (тоже через шину). Таким образом, пропускная способность шины должна быть 135 Мбайт/с, и это только для передачи видеоизображения. Но в компьютере есть еще центральный процессор и другие устройства, которые тоже должны пользоваться шиной, поэтому пропускная способность должна быть еще выше.

Максимальная частота передачи данных шины ISA — 8,33 МГц. Она способна передавать два байта за цикл, поэтому ее максимальная пропускная способность составляет 16,7 Мбайт/с. Шина EISA может передавать 4 байта за цикл. Ее пропускная способность достигает 33,3 Мбайт/с. Ясно, что ни одна из них совершенно не соответствует тому, что требуется для полноэкранного видео.

В 1990 году компания Intel разработала новую шину с гораздо более высокой пропускной способностью, чем у шины EISA. Эту шину назвали **PCI (Peripheral Component Interconnect — взаимодействие периферийных компонентов)**. Компания Intel запатентовала шину PCI и сделала все патенты всеобщим достоянием, так что любая компания могла производить периферические устройства для этой шины без каких-либо выплат за право пользования патентом. Компания Intel также сформировала промышленный консорциум Special Interest Group, который должен был заниматься дальнейшими усовершенствованиями шины PCI. Все эти действия привели к тому, что шина PCI стала чрезвычайно популярной. Фактически в каждом компьютере Intel (начиная с Pentium), а также во многих других компьютерах содержится шина PCI. Даже компания Sun выпустила версию UltraSPARC, в которой используется шина PCI (это компьютер UltraSPARC IIi). Подробно шина PCI описывается в книгах [128, 136].

Первая шина PCI передавала 32 бита за цикл и работала с частотой 33 МГц (время цикла 30 нс), общая пропускная способность составляла 133 Мбайт/с. В 1993 году появилась шина PCI 2.0, а в 1995 году — PCI 2.1. Шина PCI 2.2 подходит и для портативных компьютеров (где требуется экономия заряда батареи). Шина PCI работает с частотой 66 МГц, способна передавать 64 бита за цикл, а ее общая пропускная способность составляет 528 Мбайт/с. При такой производительности полноэкранное видеоизображение вполне достижимо (предполагается, что диск и другие устройства системы справляются со своей работой). Во всяком случае, шина PCI не будет ограничивать производительность системы.

Хотя 528 Мбайт/с — достаточно высокая скорость передачи данных, все же здесь есть некоторые проблемы. Во-первых, этого не достаточно для шины памяти. Во-вторых, эта шина не совместима со всеми старыми картами ISA. По этой причине компания Intel решила разрабатывать компьютеры с тремя и более шинами, как показано на рис. 3.47. Здесь мы видим, что центральный процессор может обмениваться информацией с основной памятью через специальную шину памяти и что

шину ISA можно связать с шиной PCI. Такая архитектура используется фактически во всех компьютерах Pentium II, поскольку она удовлетворяет всем требованиям.

Ключевыми компонентами данной архитектуры являются мосты между шинами (эти микросхемы выпускает компания Intel — отсюда такой интерес к проекту). Мост PCI связывает центральный процессор, память и шину PCI. Мост ISA связывает шину PCI с шиной ISA, а также поддерживает один или два диска IDE. Практически все системы Pentium II выпускаются с одним или несколькими свободными слотами PCI для подключения дополнительных высокоскоростных периферийных устройств и с одним или несколькими слотами ISA для подключения низкоскоростных периферийных устройств.

Преимущество системы, изображенной на рис. 3.47, состоит в том, что шина между центральным процессором и памятью имеет высокую пропускную способность, шина PCI также обладает высокой пропускной способностью и хорошо подходит для связи с быстрыми периферийными устройствами (SCSI-дисками, графическими адаптерами и т. п.), и при этом еще могут использоваться старые платы ISA. На рисунке также изображена шина USB, которую мы будем обсуждать ниже в этой главе.

Мы проиллюстрировали систему с одной шиной PCI и одной шиной ISA. На практике может использоваться и по несколько шин каждого типа. Существуют специальные мосты, которые связывают две шины PCI, поэтому в больших системах может содержаться несколько отдельных шин PCI (2 и более). В системе также может быть несколько мостов (2 и более), которые связывают шину PCI и шину ISA, что дает возможность использовать несколько шин ISA.

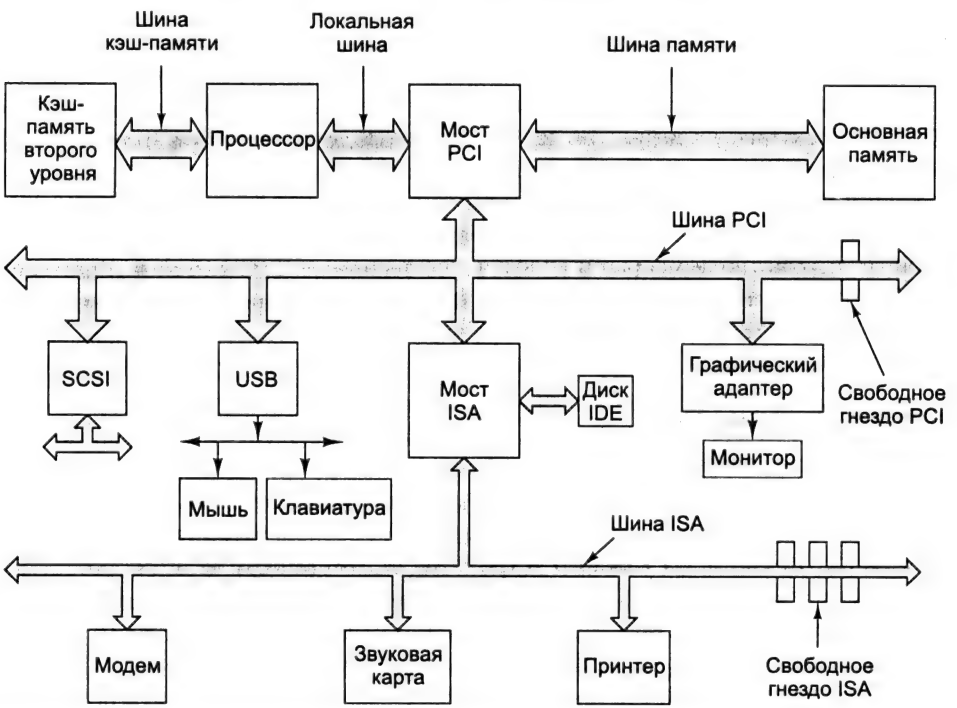


Рис. 3.47. Архитектура типичной системы Pentium II. Чем толще стрелка, обозначающая шину, тем выше пропускная способность этой шины

Было бы неплохо, если бы существовал только один тип плат PCI. К сожалению, это не так. Платы различаются по потребляемой мощности, разрядности и синхронизации. Старые компьютеры обычно используют напряжение 5 В, а новые — 3,3 В, поэтому шина PCI поддерживает и то и другое напряжение. Коннекторы одни и те же (они отличаются только двумя кусочками пластмассы, которые предназначены для того, чтобы невозможно было вставить плату на 5 В в шину PCI на 3,3 В и наоборот). К счастью, существуют и универсальные платы, которые поддерживают оба напряжения и которые можно вставить в любой слот. Платы различаются не только по мощности, но и по разрядности. Существует два типа плат: 32-битные и 64-битные. 32-битные платы содержат 120 выводов; 64-битные платы содержат те же 120 выводов плюс 64 дополнительных вывода (аналогично тому, как шина IBM PC была расширена до 16 битов, см. рис. 3.46). Шина PCI, поддерживающая 64-битные платы, может поддерживать и 32-битные, но обратное не верно. Наконец, шины PCI и соответствующие платы могут работать с частотой или 33 МГц, или 66 МГц. В обоих случаях контакты идентичны. Различие состоит в том, что один из выводов связывается либо с источником питания, либо с «землей».

Шины PCI являются синхронными, как и все шины PC, восходящие к первой модели IBM PC. Все транзакции в шине PCI осуществляются между задающим и подчиненным устройствами. Чтобы не увеличивать число выводов на плате, адресные и информационные линии объединяются. При этом достаточно 64 выводов для всей совокупности адресных и информационных сигналов, даже если PCI работает с 64-битными адресами и 64-битными данными.

Объединенные адресные и информационные выводы функционируют следующим образом. При операции считывания во время цикла 1 задающее устройство передает адрес на шину. Во время цикла 2 задающее устройство удаляет адрес и шина реверсируется таким образом, чтобы подчиненное устройство могло ее использовать. Во время цикла 3 подчиненное устройство выдает запрашиваемые данные. При операциях записи шине не нужно переключаться, поскольку задающее устройство помещает на нее и адрес, и данные. Тем не менее минимальная транзакция занимает три цикла. Если подчиненное устройство не может дать ответ в течение трех циклов, то вводится режим ожидания. Допускаются пересылки блоков неограниченного размера, а также некоторые другие типы циклов шины.

Арбитраж шины PCI

Чтобы передать по шине PCI какой-нибудь сигнал, устройство сначала должно получить к ней доступ. Шина PCI управляется централизованным арбитром, как показано на рис. 3.48. В большинстве случаев арбитр шины встраивается в один из мостов между шинами. От каждого устройства PCI к арбитру тянутся две специальные линии. Одна из них (REQ#) используется для запроса шины, а вторая (GNT#) — для получения разрешения на доступ к шине.

Чтобы сделать запрос на доступ к шине, устройство PCI (в том числе и центральный процессор) устанавливает сигнал REQ# и ждет, пока арбитр не выдаст сигнал GNT#. Если арбитр выдал сигнал GNT#, то устройство может использо-

вать шину в следующем цикле. Алгоритм, которым руководствуется арбитр, не зависит от технических характеристик шины PCI. Допустим арбитраж по кругу, по приоритету и другие схемы арбитража. Хороший арбитр должен быть справедлив, чтобы не заставлять некоторые устройства ждать целую вечность.

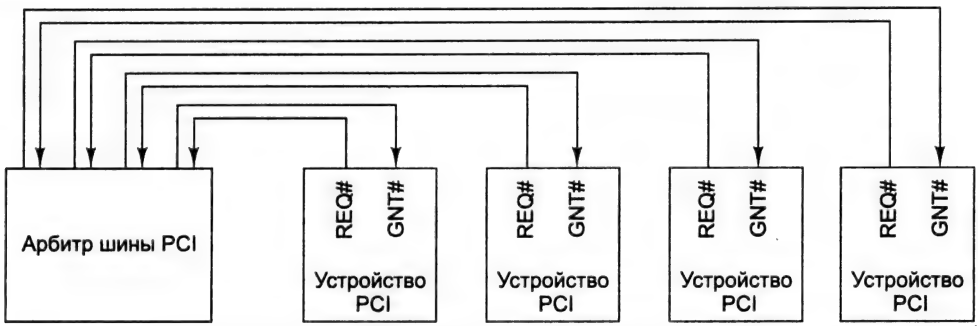


Рис. 3.48. В шине PCI используется централизованный арбитр

Шина предоставляется для одной транзакции, хотя продолжительность этой транзакции теоретически произвольна. Если устройству нужно совершить вторую транзакцию и ни одно другое устройство не запрашивает шину, оно может занять шину снова, хотя обычно между транзакциями нужно вставлять пустой цикл. Однако при особых обстоятельствах (при отсутствии конкуренции на доступ к шине) устройство может совершать последовательные транзакции без пустых циклов между ними. Если задающее устройство осуществляет очень длительную передачу, а какое-нибудь другое устройство выдало запрос на доступ к шине, арбитр может сбросить линию GNT#. Предполагается, что задающее устройство следит за линией GNT#. Если линия сбрасывается, устройство должно освободить шину в следующем цикле. Такая система позволяет осуществлять очень длинные передачи (что весьма рационально) при отсутствии конкуренции на доступ к шине, однако при этом она быстро реагирует на запросы шины, поступающие от других устройств.

Сигналы шины PCI

Шина PCI содержит ряд обязательных сигналов (табл. 3.5) и ряд факультативных сигналов (табл. 3.6). Оставшиеся выводы используются для питания, «земли» и разнообразных связанных сигналов. В столбцах «Задающее устройство» и «Подчиненное устройство» указывается, какое из устройств устанавливает сигнал при обычной транзакции. Если сигнал выдается другим устройством (например, CLK), оба столбца остаются пустыми.

Теперь давайте рассмотрим каждый сигнал шины PCI отдельно. Начнем с обязательных (32-битных) сигналов, а затем перейдем к факультативным (64-битным). Сигнал CLK запускает шину. Большинство сигналов совпадают с ним во времени. В отличие от шины ISA, в шине PCI транзакция начинается на заднем фронте сигнала CLK, то есть не в начале цикла, а в середине.

Таблица 3.5. Обязательные сигналы шины PCI

Сигнал	Количество линий	Задающее устройство	Подчиненное устройство	Комментарий
CLK	1			Тактовый генератор (33 МГц или 66 МГц)
AD	32	×	×	Объединенные адресные и информационные линии
PAR	1	×		Бит четности для адреса или данных
C/BE#	4	×		1) команда шине 2) битовый массив, который показывает, какие байты из слова нужно считать (или записать)
FRAME#	1	×		Указывает, что установлены сигналы AD и C/BE
IRDY#	1	×		При чтении: задающее устройство готово принять данные; при записи: данные находятся в шине
IDSEL	1	×		Считывание пространства конфигураций
DEVSEL#	1		×	Подчиненное устройство распознало свой адрес и ждет сигнала
TRDY#	1		×	При чтении: данные находятся на линиях AD; при записи: подчиненное устройство готово принять данные
STOP#	1		×	Подчиненное устройство требует немедленно прервать текущую транзакцию
PERR#	1			Обнаружена ошибка четности данных
SERR#	1			Обнаружена ошибка четности адреса или системная ошибка
REQ#	1			Арбитраж шины: запрос на доступ к шине
GNT#	1			Арбитраж шины: предоставление шины
RST#	1			Перезагрузка системы и всех устройств

Сигналы AD (их 32) нужны для адресов и данных (для передач по 32 бита). Обычно адрес устанавливается во время первого цикла, а данные — во время третьего. Сигнал PAR — это бит четности для сигнала AD. Сигнал C/BE# выполняет две функции. Во время первого цикла он содержит команду (считать одно слово, считать блок и т. п.). Во время второго цикла он содержит массив из 4 битов, который показывает, какие байты 32-битного слова действительны. Используя сигнал C/BE#, можно считывать 1, 2 или 3 байта из слова, а также все слово целиком.

Сигнал FRAME# устанавливается задающим устройством, чтобы начать транзакцию. Этот сигнал сообщает подчиненному устройству, что адрес и команды в данный момент действительны. При чтении одновременно с сигналом FRAME# устанавливается сигнал IRDY#. Он сообщает, что задающее устройство готово принять данные. При записи сигнал IRDY# устанавливается позже, когда данные уже находятся в шине.

Сигнал IDSEL связан с тем, что у каждого устройства PCI должно быть пространство конфигураций на 256 байтов, которое другие устройства могут считывать (установив сигнал IDSEL). Это пространство конфигураций содержит характеристики устройства. В некоторых операционных системах структура Plug-and-Play (Режим автоматического конфигурирования) использует это пространство конфигураций, чтобы распознать, какие устройства подключены к шине.

А теперь рассмотрим сигналы, которые устанавливаются подчиненным устройством. Сигнал DEVSEL# означает, что подчиненное устройство распознало свой адрес на линиях AD и готово участвовать в транзакции. Если сигнал DEVSEL# не поступает в течение определенного промежутка времени, задающее устройство предполагает, что подчиненное устройство, к которому направлено обращение, либо отсутствует, либо неисправно.

Следующий сигнал — TRDY#. Его подчиненное устройство устанавливает при чтении, чтобы сообщить, что данные находятся на линиях AD, и при записи, чтобы сообщить, что оно готово принять данные.

Следующие три сигнала нужны для сообщения об ошибках. Один из них, сигнал STOP#, устанавливается подчиненным устройством, если произошла какая-нибудь неполадка и нужно прервать текущую транзакцию. Следующий сигнал, PERR#, используется для сообщения об ошибке четности в данных в предыдущем цикле. Для чтения этот сигнал устанавливается задающим устройством, для записи — подчиненным устройством. Необходимые действия должно предпринимать устройство, получившее этот сигнал. Наконец, сигнал SERR# нужен для сообщения об адресных и системных ошибках.

Таблица 3.6. Факультативные сигналы шины PCI

Сигнал	Количество линий	Задающее устройство	Подчиненное устройство	Комментарий
REQ64#	1	×		Запрос на осуществление 64-битной транзакции
ACK64#	1		×	Разрешение 64-битной транзакции
AD	32	×		Дополнительные 32 бита адреса или данных
PAR64	1	×		Проверка четности для дополнительных 32 битов адреса или данных
C/BE#	4	×		Дополнительные 4 бита для указания, какие байты из слова нужно считать (или записать)
LOCK	1	×		В многопроцессорных системах: блокировка шины при осуществлении транзакции одним из процессоров
SBO#	1			Обращение к кэш-памяти другого процессора
SDONE	1			Отслеживание адресов, по которым произошли изменения, завершено.
INTx	4			Запрос прерывания
JTAG	5			Сигналы тестирования IEEE 1149.1 JTAG
M66EN	1			Сигнал связывается с источником питания или с «землей» (66 МГц или 33 МГц)

Сигналы REQ# и GNT# предназначены для арбитража шины. Они устанавливаются не тем устройством, которое является задающим в данный момент, а тем, которому нужно стать задающим. Последний обязательный сигнал, RST#, используется для перезагрузки системы, которая происходит, либо если пользователь нажмет кнопку RESET, либо если какое-нибудь системное устройство обнаружит фатальную ошибку. После установки этого сигнала компьютер перезагружается.

Перейдем к факультативным сигналам, большинство из которых связано с расширением разрядности с 32 до 64 битов. Сигналы REQ64# и ACK 64# позволяют задающему устройству попросить разрешение осуществить 64-битную транзакцию, а подчиненному устройству принять эту транзакцию. Сигналы AD, PAR64 и C/BE# являются расширениями соответствующих 32-битных сигналов.

Следующие три сигнала не связаны с противопоставлением 32 бита — 64 бита. Они имеют отношение к многопроцессорным системам. Не все платы PCI поддерживают такие системы, поэтому эти сигналы являются факультативными. Сигнал LOCK позволяет блокировать шину для параллельных транзакций. Следующие два сигнала связаны с отслеживанием всех адресов, по которым происходит изменение данных. Подобное отслеживание необходимо для того, чтобы сохранить непротиворечивость кэш-памяти различных процессоров.

Сигналы INT_x нужны для запроса прерываний. Плата PCI может содержать до четырех логических устройств, каждое из которых имеет собственную линию запроса прерывания. Сигналы JTAG предназначены для процедуры тестирования IEEE 1149.1 JTAG. Наконец, сигнал M66EN связывается либо с источником питания, либо с «землей», что определяет тактовую частоту. Она не должна меняться во время работы системы.

Транзакции шины PCI

Шина PCI в действительности очень проста. Чтобы лучше понять это, рассмотрим временную диаграмму на рис. 3.49. Здесь мы видим транзакцию чтения, за ней следует пустой цикл и транзакция записи, которая осуществляется тем же задающим устройством.

Во время цикла T_1 на заднем фронте синхронизирующего сигнала задающее устройство помещает адрес на линии AD и команду на линии C/BE#. Затем задающее устройство устанавливает сигнал FRAME#, чтобы начать транзакцию.

Во время цикла T_2 задающее устройство переключает шину, чтобы подчиненное устройство могло воспользоваться ею во время цикла T_3. Задающее устройство также изменяет сигнал C/BE#, чтобы указать, какие байты в слове ему нужно считать.

Во время цикла T_3 подчиненное устройство устанавливает сигнал DEVSEL#. Этот сигнал сообщает задающему устройству, что подчиненное устройство получило адрес и собирается ответить. Подчиненное устройство также помещает данные на линии AD и выдает сигнал TRDY#, который сообщает задающему устройству о данном действии. Если подчиненное устройство не может ответить быстро, оно не снимает сигнал DEVSEL#, который сообщает о его присутствии, но при этом не устанавливает сигнал TRDY# до тех пор, пока не сможет передать данные. При такой процедуре вводится один или несколько периодов ожидания.

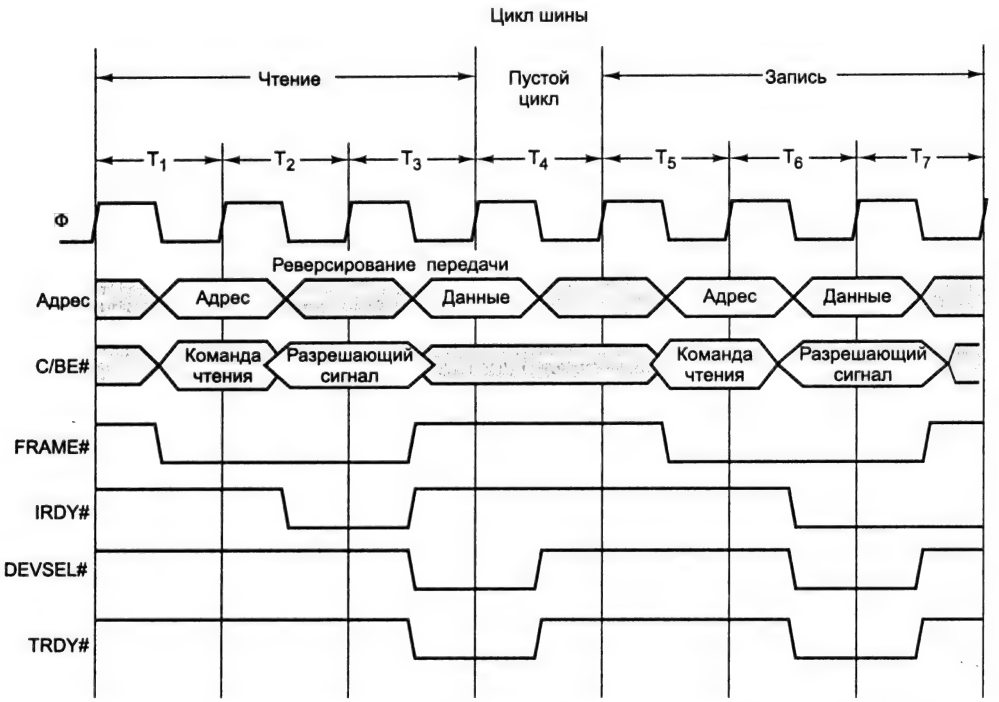


Рис. 3.49. Примеры 32-битных транзакций в шине PCI. Во время первых трех циклов происходит операция чтения, затем идет пустой цикл, а следующие три цикла — операция записи

В нашем примере (часто это бывает и в действительности) следующий цикл пустой. Мы видим, что в цикле T_5 то же самое задающее устройство инициирует процесс записи. Сначала оно, как обычно, помещает адрес и команду на шину. В следующем цикле оно выдает данные. Поскольку линиями AD управляет одно и то же устройство, цикл реверсирования передачи не требуется. В цикле T_7 память принимает данные.

Шина USB

Шина PCI очень хорошо подходит для подсоединения высокоскоростных периферических устройств, но использовать интерфейс PCI для низкоскоростных устройств ввода-вывода (например, мыши и клавиатуры) было бы слишком дорого. Изначально каждое стандартное устройство ввода-вывода подсоединялось к компьютеру особым образом, при этом для добавления новых устройств использовались свободные слоты ISA и PCI. К сожалению, такая схема имеет некоторые недостатки. Например, каждое новое устройство ввода-вывода часто снабжено собственной платой ISA или PCI. Пользователь при этом должен сам установить переключатели и перемычки на плате и увериться, что установленные параметры не конфликтуют с другими платами. Затем пользователь должен открыть системный блок, аккуратно вставить плату, закрыть системный блок, а затем включить компьютер. Для многих этот процесс очень сложен и часто приводит к ошибкам.

Кроме того, число слотов ISA и PCI очень мало (обычно их два или три). Платы Plug and Play исключают установку переключателей, но пользователь все равно должен открывать компьютер и вставлять туда плату. К тому же количество слотов шины ограничено.

В середине 90-х годов представители семи компаний (Compaq, DEC, IBM, Intel, Microsoft, NEC и Nothern Telecom) собрались вместе, чтобы разработать шину, оптимально подходящую для подсоединения низкоскоростных устройств. Потом к ним примкнули сотни других компаний. Результатом их работы стала шина **USB (Universal Serial Bus — универсальная последовательная шина)**, которая сейчас широко используется в персональных компьютерах. Она подробно описана в книгах [7, 144].

Некоторые требования, изначально составляющие основу проекта:

1. Пользователи не должны устанавливать переключатели и перемычки на платах и устройствах.
2. Пользователи не должны открывать компьютер, чтобы установить новые устройства ввода-вывода.
3. Должен существовать только один тип кабеля, подходящий для подсоединения всех устройств.
4. Устройства ввода-вывода должны получать питание через кабель.
5. Необходима возможность подсоединения к одному компьютеру до 127 устройств.
6. Система должна поддерживать устройства реального времени (например, звук, телефон).
7. Должна быть возможность устанавливать устройства во время работы компьютера.
8. Должна отсутствовать необходимость перезагружать компьютер после установки нового устройства.
9. Производство новой шины и устройств ввода-вывода для нее не должно требовать больших затрат.

Шина USB удовлетворяет всем этим условиям. Она разработана для низкоскоростных устройств (клавиатур, мышей, фотоаппаратов, сканеров, цифровых телефонов и т. д.). Общая пропускная способность шины составляет 1,5 Мбайт/с. Этого достаточно для большинства таких устройств. Предел был выбран для того, чтобы снизить стоимость шины.

Шина USB состоит из **центрального хаба**[1], который вставляется в разъем главной шины (см. рис. 3.47). Этот центральный хаб (часто называемый корневым концентратором) содержит разъемы для кабелей, которые могут подсоединяться к устройствам ввода-вывода или к дополнительным хабам, чтобы обеспечить большее количество разъемов. Таким образом, топология шины USB представляет собой дерево с корнем в центральном хабе, который находится внутри компьютера. Коннекторы кабеля со стороны устройства отличаются от коннекторов со стороны хаба, чтобы пользователь случайно не подсоединил кабель другой стороной.

[1] От англ. *hub* – концентратор. — *Примеч. пер.*

Кабель состоит из четырех проводов: два из них предназначены для передачи данных, один — для источника питания (+5 В) и один — для «земли». Система передает 0 изменением напряжения, а 1 — отсутствием изменения напряжения, поэтому длинная последовательность нулей порождает поток регулярных импульсов.

Когда подсоединяется новое устройство ввода-вывода, центральный хаб (концентратор) распознает это и прерывает работу операционной системы. Затем операционная система запрашивает новое устройство, что оно собой представляет и какая пропускная способность шины для него требуется. Если операционная система решает, что для этого устройства пропускной способности достаточно, она приписывает ему уникальный адрес (1–127) и загружает этот адрес и другую информацию в регистры конфигурации внутри устройства. Таким образом, новые устройства могут подсоединяться «на лету», при этом пользователю не нужно устанавливать новые платы ISA или PCI. Неинициализированные платы начинаются с адреса 0, поэтому к ним можно обращаться. Многие устройства снабжены встроенными сетевыми концентраторами для дополнительных устройств. Например, монитор может содержать два хаба для правой и левой колонок.

Шина USB представляет собой ряд каналов от центрального хаба к устройствам ввода-вывода. Каждое устройство может разбить свой канал максимум на 16 подканалов для различных типов данных (например, аудио и видео). В каждом канале или подканале данные перемещаются от центрального концентратора к устройству или обратно. Между двумя устройствами ввода-вывода обмена информацией не происходит.

Ровно через каждую миллисекунду (±0,05 мс) центральный концентратор передает новый кадр, чтобы синхронизировать все устройства во времени. Кадр состоит из пакетов, первый из которых передается от концентратора к устройству. Следующие пакеты кадра могут передаваться в том же направлении, а могут и в противоположном (от устройства к хабу). На рис. 3.50 показаны четыре последовательных кадра.

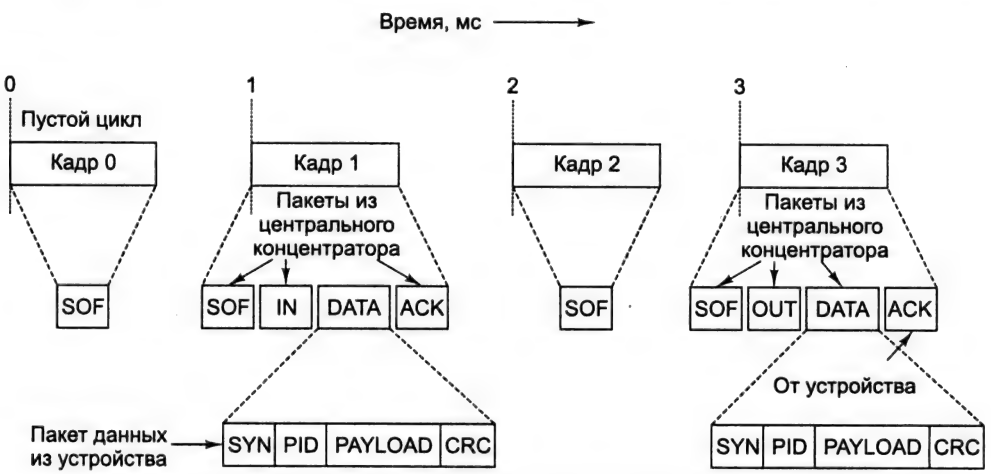


Рис. 3.50. Центральный концентратор шины USB передает кадры каждую миллисекунду

Рассмотрим рис. 3.50. В кадрах 0 и 2 не происходит никаких действий, поэтому в них содержится только пакет SOF (Start of Frame — начало кадра). Этот пакет всегда посылается всем устройствам. Кадр 1 — упорядоченный опрос (например, сканеру посылается запрос на передачу битов сканированного им изображения). Кадр 3 состоит из отсылки данных какому-нибудь устройству (например, принтеру).

Шина USB поддерживает 4 типа кадров: кадры управления, изохронные кадры, кадры передачи больших массивов данных и кадры прерывания. Кадры управления используются для конфигурации устройств, передачи команд устройствам и запросов об их состоянии. Изохронные кадры предназначены для устройств реального времени (микрофонов, акустических систем и телефонов), которые должны принимать и посылать данные через равные временные интервалы. Задержки хорошо прогнозируются, но в случае ошибки такие устройства не производят повторной передачи. Кадры следующего типа используются для передач большого объема от устройств и к устройствам без требований реального времени (например, принтеров). Наконец, кадры последнего типа нужны для того, чтобы осуществлять прерывания, поскольку шина USB не поддерживает прерывания. Например, вместо того чтобы вызывать прерывание всякий раз, когда происходит нажатие клавиши, операционная система может вызывать прерывания каждые 50 млс и «собирать» все задержанные нажатия клавиш.

Кадр состоит из одного или нескольких пакетов. Пакеты могут посылаться в обоих направлениях. Существует четыре типа пакетов: маркеры, пакеты данных, пакеты квитирования и специальные пакеты. Маркеры передаются от концентратора к устройству и предназначены для управления системой. Пакеты SOF, IN и OUT на рис. 3.50 — маркеры. Пакет SOF (Start of Frame — начало кадра) является первым в любом кадре и показывает начало кадра. Если никаких действий выполнять не нужно, пакет SOF единственный в кадре. Пакет IN — это запрос. Этот пакет требует, чтобы устройство выдало определенные данные. Поля в пакете IN содержат информацию, какой именно канал запрашивается, и таким образом устройство определяет, какие именно данные выдавать (если оно обращается с несколькими потоками данных). Пакет OUT объявляет, что далее последует передача данных для устройства. Последний тип маркера, SETUP (он не показан на рисунке), используется для конфигурации.

Кроме маркеров существует еще три типа пакетов. Это пакеты DATA (используются для передачи 64 байтов информации в обоих направлениях), пакеты квитирования и специальные пакеты. Формат пакета данных показан на рис. 3.50. Он состоит из 8-битного поля синхронизации, 8-битного указателя типа пакета (PID), полезной нагрузки и 16-битного **CRC (Cyclic Redunduncy Code — циклический избыточный код)** для обнаружения ошибок. Есть три типа пакетов квитирования: ACK (предыдущий пакет данных был принят правильно), NAC (найдена ошибка CRC) и STALL (подождите, пожалуйста, я сейчас занят).

А теперь давайте снова посмотрим на рис. 3.50. Центральный концентратор должен отсылать новый кадр каждую миллисекунду, даже если не происходит никаких действий. Кадры 0 и 2 содержат только один пакет SOF, что говорит о том, что ничего не происходит. Кадр 1 представляет собой опрос, поэтому он начинается с пакетов SOF и IN, которые передаются от компьютера к устройству ввода-вывода, а затем следует пакет DATA от устройства к компьютеру. Пакет ACK сообщает

устройству, что данные были получены без ошибок. В случае ошибки устройство получает пакет NACK, после чего данные передаются заново (отметим, что изохронные данные повторно не передаются). Кадр 3 похож по структуре на кадр 1, но в нем поток данных направлен от компьютера к устройству.

Средства сопряжения

Обычная компьютерная система малого или среднего размера состоит из микросхемы процессора, микросхем памяти и нескольких контроллеров ввода-вывода. Все эти микросхемы соединены шиной. Мы уже рассмотрели память, центральные процессоры и шины. Теперь настало время изучить микросхемы ввода-вывода. Именно через эти микросхемы компьютер обменивается информацией с внешними устройствами.

Микросхемы ввода-вывода

В настоящее время существует множество различных микросхем ввода-вывода. Новые микросхемы появляются постоянно. Из наиболее распространенных можно назвать UART, USART, контроллеры CRT (CRT — электронно-лучевая трубка), дисковые контроллеры и PIO. **UART (Universal Asynchronous Receiver Transmitter — универсальный асинхронный приемопередатчик)** — это микросхема, которая может считывать байт из шины данных и передавать этот байт по битам на линию последовательной передачи к терминалу или получать данные от терминала. Скорость работы микросхем UART различна: от 50 до 19 200 бит/с; ширина знака от 5 до 8 битов; 1, 1,5 или 2 стоповых бита. Микросхема может обеспечивать проверку на четность или на нечетность, контроль по четности может также отсутствовать, все это находится под управлением программ. Микросхема **USART (Universal Synchronous Asynchronous Receiver Transmitter — универсальный синхронно-асинхронный приемопередатчик)** может осуществлять синхронную передачу, используя ряд протоколов. Она также выполняет все функции UART. Поскольку микросхемы UART мы уже рассматривали в главе 2, сейчас в качестве примера микросхемы ввода-вывода мы возьмем параллельный интерфейс.

Микросхемы PIO

Типичным примером микросхемы **PIO (Parallel Input/Output — параллельный ввод-вывод)** является Intel 8255A (рис. 3.51). Она содержит 24 линии ввода-вывода и может сопрягаться с любыми устройствами, совместимыми с TTL-схемами (например, клавиатурами, переключателями, индикаторами, принтерами). Программа центрального процессора может записать 0 или 1 на любую линию или считать входное состояние любой линии, обеспечивая высокую гибкость. Микросхема PIO часто заменяет целую плату с микросхемами МИС и СИС (особенно во встроенных системах).

Центральный процессор может конфигурировать микросхему 8255A различными способами, загружая регистры состояния микросхемы, и мы остановимся на

некоторых наиболее простых режимах работы. Можно представить данную микросхему в виде трех 8-битных портов A, B и C. С каждым портом связан 8-битный регистр. Чтобы установить линии на порт, центральный процессор записывает 8-битное число в соответствующий регистр, и это 8-битное число появляется на выходных линиях и остается там до тех пор, пока регистр не будет перезаписан. Чтобы использовать порт для входа, центральный процессор просто считывает соответствующий регистр.



Рис. 3.51. Микросхема 8255A

Другие режимы работы предусматривают квитирование связи с внешними устройствами. Например, чтобы передать данные устройству, микросхема 8255A может представить данные на порт вывода и подождать, пока устройство не выдаст сигнал о том, что данные получены и можно посылать еще. В данную микросхему включены необходимые логические схемы для фиксирования таких импульсов и передачи их центральному процессору.

Из рис. 3.51 мы видим, что помимо 24 выводов для трех портов микросхема 8255A содержит восемь линий, непосредственно связанных с шиной данных, линию выбора элемента памяти, линии чтения и записи, две адресные линии и линию для переустановки микросхемы. Две адресные линии выбирают один из четырех внутренних регистров, три из которых соответствуют портам A, B и C. Четвертый регистр — регистр состояния. Он определяет, какие порты используются для входа, а какие для выхода, а также выполняет некоторые другие функции. Обычно две адресные линии соединяются с двумя младшими битами адресной шины.

Декодирование адреса

До настоящего момента мы не останавливались подробно на том, как происходит выбор микросхемы памяти или устройства ввода-вывода. Пришло время это узнать. Рассмотрим простой 16-битный встроенный компьютер, состоящий из центрального процессора, стираемого программируемого ПЗУ объемом 2K×8 байт для хранения программы, ОЗУ объемом 2K×8 байт для хранения данных и микросхемы PIO. Такая небольшая система может встраиваться в дешевую игрушку или простой прибор. Вместо стираемого программируемого ПЗУ может использоваться обычное ПЗУ.

Микросхема PIO может быть выбрана одним из двух способов: как устройство ввода-вывода или как часть памяти. Если микросхема нам нужна в качестве

устройства ввода-вывода, мы должны выбрать ее, используя внешнюю линию шины, которая показывает, что мы обращаемся к устройству ввода-вывода, а не к памяти. Если мы применяем другой подход, так называемый **ввод-вывод с распределением памяти**, мы должны присвоить микросхеме 4 байта памяти для трех портов и регистра управления. Наш выбор в какой-то степени произволен. Мы выбираем ввод-вывод с распределением памяти, поскольку этот метод наглядно иллюстрирует некоторые интересные проблемы сопряжения.

Стираемому программируемому ПЗУ требуется 2 К адресного пространства, ОЗУ требуется также 2 К адресного пространства, а микросхеме PIO нужно 4 байта. Поскольку в нашем примере адресное пространство составляет 64 К, мы должны выбрать, где поместить данные три устройства. Один из возможных вариантов показан на рис. 3.52. Стираемое программируемое ПЗУ занимает адреса до 2 К, ОЗУ занимает адреса от 32 К до 34 К, а PIO — 4 старших байта адресного пространства, от 65532 до 65535. С точки зрения программиста не важно, какие именно адреса использовать, однако для сопряжения это имеет большое значение. Если бы мы обращались к PIO через пространство ввода-вывода, нам не потребовались бы адреса памяти (зато понадобились бы четыре адреса пространства ввода-вывода).

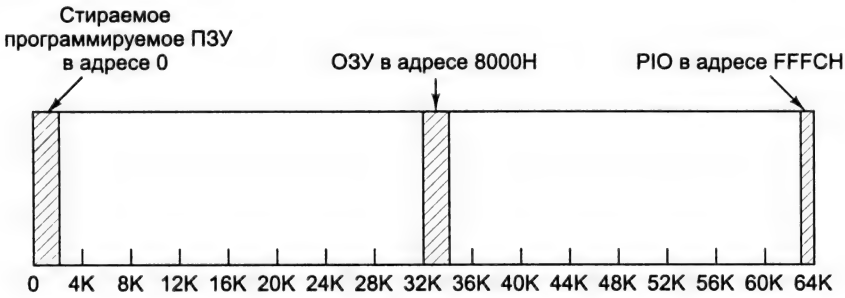


Рис. 3.52. Расположение стираемого ПЗУ, ОЗУ и PIO на адресном пространстве в 64 К

При таком распределении адресов (рис. 3.52) стираемое ПЗУ нужно выбирать с помощью 16-битного адреса памяти 00000xxxxxxxxxxx (в двоичной системе). Другими словами, любой адрес, у которого пять старших битов равны 0, попадает в область памяти до 2 К и, следовательно, в стираемое ПЗУ. Таким образом, сигнал выбора стираемого ПЗУ можно связать с 5-разрядным компаратором, у которого один из входов всегда будет соединен с 00000.

Чтобы достичь того же результата, лучше было бы использовать пятивходовый вентиль ИЛИ, у которого пять входов связаны с адресными линиями от A11 до A15. Выходной сигнал будет равен 0 тогда и только тогда, когда все пять линий равны 0. В этом случае устанавливается сигнал $\overline{CS}$. К сожалению, в стандартных сериях МИС не существует пятивходовых вентилей ИЛИ. Однако мы можем использовать восьмивходовый вентиль НЕ-ИЛИ. Заземлив три входа и инвертировав выход, мы можем получить нужный нам сигнал (рис. 3.53, *а*). Схемы МИС стоят очень дешево, поэтому неэффективное использование одной из них вполне допустимо. По соглашению неиспользуемые входы на схемах не показываются.

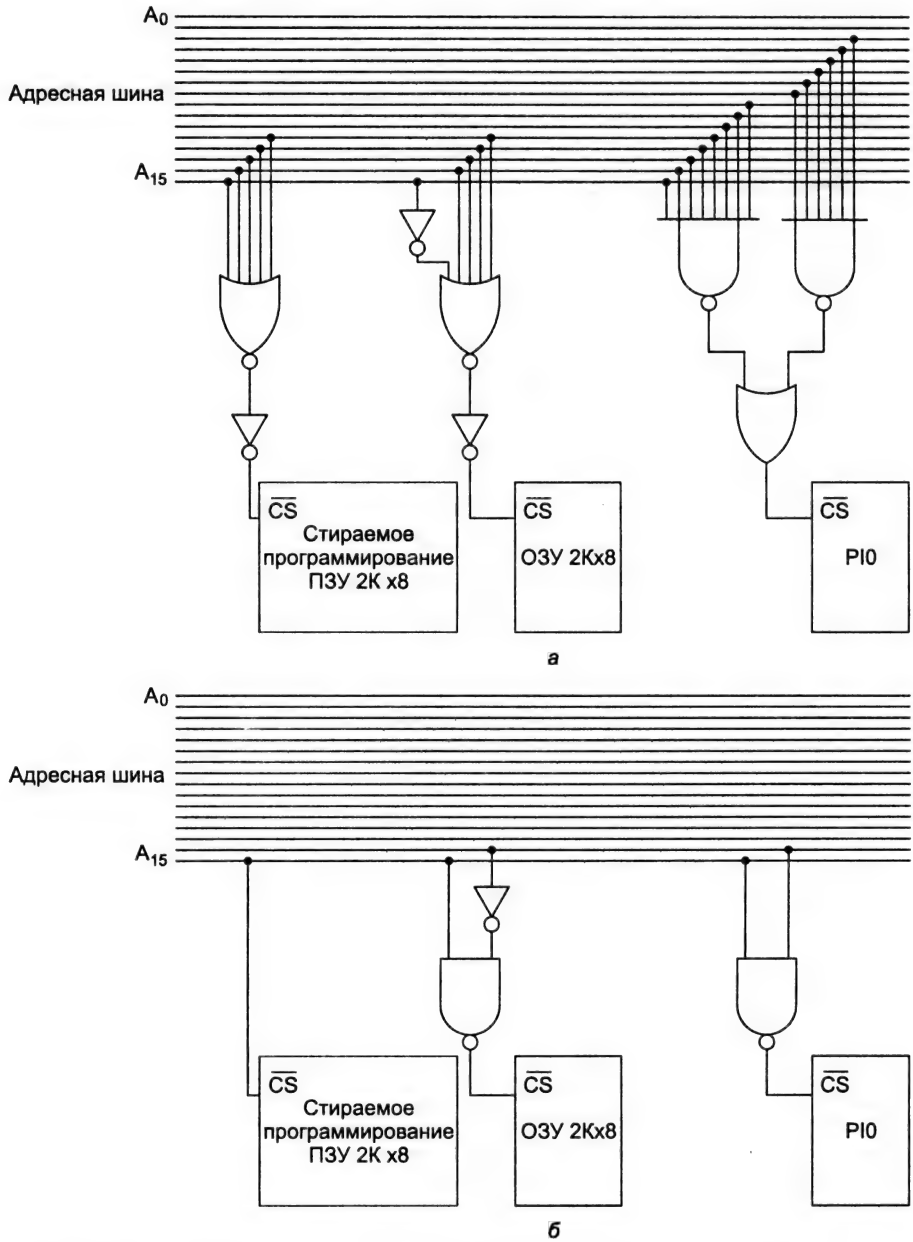


Рис. 3.53. Полное декодирование адреса (*а*); частичное декодирование адреса (*б*)

Тот же принцип можно применить и для ОЗУ. Однако ОЗУ должно отвечать на бинарные адреса типа 10000xxxxxxxxxx, поэтому необходим дополнительный инвертор (он показан на схеме). Декодирование адреса микросхемы PIO несколько сложнее, поскольку данная микросхема выбирается с помощью 4 адресов типа 1111111111111xx. Один из возможных вариантов схемы, которая устанавливает

сигнал CS только в том случае, если на адресной шине появляется адрес данного типа, показан на рис. 3.53. Здесь используются два восьмивходовых вентиля НЕ-И, которые соединяются с вентилем ИЛИ. Чтобы сконструировать схему декодирования адреса, изображенную на рис. 3.53, *а*, требуется шесть микросхем МИС: четыре восьмивходовые микросхемы, вентиль ИЛИ и микросхема с тремя инверторами.

Если компьютер состоит только из центрального процессора, двух микросхем памяти и PIO, мы можем сильно упростить процесс декодирования адреса. Дело в том, что у всех адресов стираемого ПЗУ и только у адресов стираемого ПЗУ старший разряд A15 всегда равен 0. Следовательно, мы можем просто связать сигнал $\overline{\text{CS}}$ с линией A15, как показано на рис. 3.53, *б*.

Теперь решение поместить ОЗУ в адрес 8000H кажется не таким уж произвольным. Отметим, что в ОЗУ попадают адреса типа 10xxxxxxxxxxxxxx, поэтому для декодирования достаточно двух битов. Точно так же, любой адрес, начинающийся с 11, является адресом PIO. Полная логика декодирования состоит из двух вентилей НЕ-И и инвертора. Поскольку инвертор можно сделать из вентиля НЕ-И, связав два входа вместе, одного счетверенного вентиля НЕ-И более чем достаточно.

Логика декодирования адреса, показанная на рис. 3.53, *б*, называется **частичным декодированием адреса**, поскольку в данном случае полные адреса не используются. При таком декодировании считывание из адресов 0001000000000000, 0001100000000000 и 0010000000000000 будет давать один и тот же результат. В действительности любой адрес в нижней половине адресного пространства будет выбирать стираемое ПЗУ. Поскольку дополнительные адреса не используются, в этом нет ничего ужасного, но при разработке компьютера, который будет расширяться в будущем (в случае с игрушками это маловероятно), следует избегать частичного декодирования, поскольку оно сильно ограничивает адресное пространство.

Можно применять и другую технологию декодирования адреса — технологию с использованием декодера (см. рис. 3.12). Связав три входа с тремя адресными линиями самых старших разрядов, мы получаем восемь выходов, которые соответствуют адресам в первом отрезке 8 K, втором отрезке 8 K и т. д. В компьютере, содержащем 8 микросхем ОЗУ по 8 K×8 байт, полное декодирование осуществляет одна такая микросхема. Если компьютер содержит 8 микросхем памяти по 2 K×8 байт, для декодирования также достаточно одного декодера, при условии что каждая микросхема памяти занимает отдельный участок адресного пространства в 8 K. (Вспомните наше замечание о том, что расположение микросхем памяти и устройств ввода-вывода внутри адресного пространства имеет значение.)

Краткое содержание главы

Компьютеры конструируются из интегральных схем, содержащих крошечные переключатели, которые называются вентилями. Обычно используются вентили И, ИЛИ, НЕ-И, НЕ-ИЛИ и НЕ. Комбинируя отдельные вентили, можно построить простые схемы.

Более сложными схемами являются мультиплексоры, демультиплексоры, кодеры, декодеры, схемы сдвига и АЛУ. С помощью программируемой логической матрицы можно запрограммировать произвольные булевы функции. Если требу-

ется много булевых функций, программируемые логические матрицы обычно более эффективны, чем другие средства. Законы булевой алгебры используются для преобразования схем из одной формы в другую. Во многих случаях таким способом можно произвести более экономичные схемы.

Арифметические действия в компьютерах осуществляются сумматорами. Одноразрядный полный сумматор можно сконструировать из двух полусумматоров. Чтобы построить сумматор для многоразрядных слов, полные сумматоры нужно соединить таким образом, чтобы выход переноса каждого сумматора передавался его левому соседу.

Статическая память состоит из защелок и триггеров, каждый из которых может хранить один бит информации. Их можно объединять и получать восьмиразрядные триггеры и защелки либо законченную память для хранения слов. Существуют различные типы памяти: ОЗУ, ПЗУ, программируемое ПЗУ, стираемое ПЗУ, электронно-перепрограммируемое ПЗУ и флэш-память. Статическое ОЗУ не нужно обновлять: оно хранит информацию, пока включен компьютер. Динамическое ОЗУ, напротив, нужно периодически обновлять, чтобы предотвратить утечку информации.

Компоненты компьютерной системы соединяются шинами. Большинство выводов обычного центрального процессора (хотя не все) запускают одну линию шины. Линии шины можно подразделить на адресные, информационные и линии управления. Синхронные шины запускаются задающим генератором. В асинхронных шинах для согласования работы задающего и подчиненного устройств используется система полного квитирования.

Pentium II представляет собой пример современного процессора. Системы с таким процессором включают в себя шину памяти, шину PCI, шину ISA и шину USB. Шина PCI может передавать за один раз 64 бита информации с частотой 66 МГц. Этого вполне достаточно практически для всех периферических устройств, но не для памяти.

Переключатели, индикаторы, принтеры и многие другие устройства ввода-вывода можно связать с компьютером, используя микросхемы ввода-вывода (например, 8255A). Эти микросхемы по желанию можно сделать частью пространства ввода-вывода или частью пространства памяти. Выбор микросхемы может происходить с помощью полного или частичного декодирования адреса в зависимости от того, какие задачи выполняет компьютер.

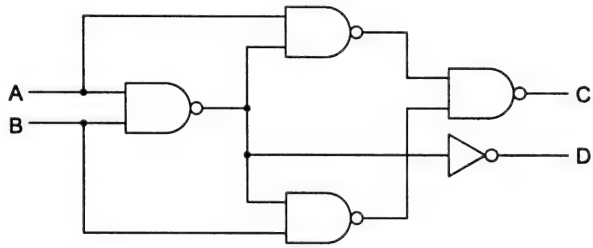
Вопросы и задания

1. Логик заезжает в закусочную и говорит: «Дайте мне, пожалуйста, гамбургер или хот-дог и картофель фри». К несчастью, повар не закончил и шести классов и не знает (да и не хочет знать), какая из двух логических операций, «и» или «или», имеет приоритет над другой. Он считает, что в данном случае допустима любая интерпретация. А какие из нижеперечисленных интерпретаций этого высказывания действительно допустимы? (Отметим, что «или» означает «исключающее ИЛИ»).

1. Только гамбургер.
2. Только хот-дог.
3. Только картофель фри.
4. Хот-дог и картофель фри.
5. Гамбургер и картофель фри.
6. Хот-дог и гамбургер.
7. Все три.
8. Ничего — логик голодает, потому что он слишком умный.

2. Миссионер, заблудившийся в Южной Калифорнии, остановился на развилке дороги. Он знает, что в этом районе обитают две мотоциклетные банды. Одна из них всегда говорит правду, а другая всегда лжет. Он хочет узнать, какая дорога ведет в Диснейленд. Какой вопрос он должен задать?
3. Существует 4 булевы функции от одной переменной и 16 функций от двух переменных. Сколько существует функций от трех переменных? А от n переменных?
4. Используя таблицу истинности, покажите, что P=(P И Q) ИЛИ (P И НЕ Q).
5. Покажите, как можно воплотить функцию И, используя два вентиля НЕ-И.
6. Используя закон Де Моргана, найдите дополнение от $A\overline{B}$.
7. Используя мультиплексор с тремя переменными, изображенный на рис. 3.11, реализуйте функцию, значение которой равно 1 тогда и только тогда, когда нечетное число входных сигналов равно 1.
8. Мультиплексор с тремя переменными, изображенный на рис. 3.11, в действительности способен вычислять произвольную функцию от четырех логических переменных. Опишите, как это происходит, и нарисуйте логическую схему для функции, которая принимает значение 0, если слово, соответствующее строке таблицы истинности, содержит четное число букв, и 1, если оно содержит нечетное число букв (например, 0000 = нуль = четыре буквы → 0; 0010 = два = три буквы → 1; 0111 = семь = четыре буквы → 0; 1101 = тринадцать = десять букв → 0). *Подсказка*: назовем четвертую входную переменную D. Тогда восемь входных линий можно связать с V_{cc}, «землей», D или $\overline{D}$.
9. Нарисуйте логическую схему 2-разрядного демультиплексора, у которого сигнал на единственной входной линии направляется к одной из четырех выходных линий в зависимости от значений двух линий управления.
10. Нарисуйте логическую схему 2-разрядного кодера, который содержит 4 входные и 2 выходные линии. Ровно одна из входных линий всегда равна 1. Двухразрядное двоичное число на 2 выходных линиях показывает, какая именно входная линия равна 1.
11. Перерисуйте программируемую логическую матрицу, изображенную на рис. 3.14. Покажите, как на ней можно реализовать логическую функцию большинства (см. рис. 3.3). Обязательно покажите, какие из потенциально возможных связей используются в первой и второй матрице.

12. Что делает данная схема?



13. Обычная схема СИС представляет собой 4-разрядный сумматор. Четыре такие микросхемы можно связать вместе и получить 16-разрядный сумматор. Как вы думаете, сколько выводов должен содержать каждый 4-разрядный сумматор? Почему?

14. n-разрядный сумматор можно получить путем каскадирования n полных сумматоров, причем перенос в стадию i, который мы будем обозначать Ci, получается из результата вычислений на стадии i–1. Перенос в стадию 0 (C_0) равен 0. Если вычисление суммы и переноса составляет на каждой стадии T нс, то перенос в стадию i будет вычислен только через iT нс после начала суммирования. При большом n до вычисления переноса в последнюю стадию может пройти очень много времени. Разработайте сумматор, который работает быстрее. *Подсказка*: каждый перенос C_i можно выразить через операнды (биты) A_{i-1} и B_{i-1}, так же как и перенос C_{i-1}. Используя это соответствие, можно выразить C_i как функцию от входов на стадии от 0 до i–1, так что все переносы можно будет генерировать одновременно.

15. Если все вентили на рис. 3.18 имеют задержку на прохождение сигнала 10 нс, а все прочие задержки не учитываются, сколько потребуется времени (минимум) для получения достоверного выходного сигнала?

16. АЛУ, изображенное на рис. 3.19, способно выполнять сложение 8-разрядных двоичных чисел. Может ли оно выполнять вычитание двоичных чисел? Если да, то объясните, как. Если нет, преобразуйте схему таким образом, чтобы она могла вычитать.

17. Иногда бывает нужно, чтобы 8-разрядное АЛУ (см., например, рис. 3.19) выдавало на выходе константу –1. Предложите два различных способа того, как это можно сделать. Для каждого способа определите значения шести сигналов управления.

18. 16-разрядное АЛУ конструируется из 16 одноразрядных АЛУ, каждое из которых тратит на суммирование 10 нс. Если задержка на прохождение сигнала от одного АЛУ к другому составляет 1 нс, сколько времени потребуется для получения конечного результата?

19. Каково состояние покоя входов S и R SR-защелки, построенной из двух вентилей НЕ-И?

20. Схема на рис. 3.25 представляет собой триггер, который запускается на нарастающем фронте синхронизирующего сигнала. Преобразуйте эту схему

так, чтобы получить триггер, который запускается на заднем фронте синхронизирующего сигнала.

21. Вы консультируете неопытных производителей микросхем МИС. Один из ваших клиентов предложил выпустить микросхему, содержащую четыре D-триггера, каждый из которых имеет выходы Q и $\overline{Q}$ по требованию потенциального важного покупателя. В данном проекте все 4 синхронизирующих сигнала объединены (также по требованию). Входов предварительной установки и очистки у схемы нет. Ваша задача — дать профессиональную оценку этой разработки.
22. В памяти 4×3, изображенной на рис. 3.28, используется 22 вентиля И и три вентиля ИЛИ. Сколько потребуется вентилей каждого из двух типов, если схему расширить до размеров 256×8?
23. С увеличением объема памяти, помещаемой на одну микросхему, число выводов, необходимых для обращения к этой памяти, также увеличивается. Иметь большое количество адресных выводов на микросхеме довольно неудобно. Придумайте способ обращения к 2^n словам памяти при наличии меньшего количества выводов, чем n.
24. В компьютере с 32-битной шиной данных используются динамические ОЗУ размером 1 М×1. Каков минимальный объем памяти (в байтах), который может содержаться в этом компьютере?
25. Посмотрите на временную диаграмму на рис. 3.34. Предположим, что вы замедлили задающий генератор до периода в 40 нс вместо 25 нс, но временные ограничения сохранились без изменений. Сколько времени в худшем случае будет у памяти на то, чтобы передать данные на шину во время T_3 после того, как был установлен сигнал $\overline{\text{MREQ}}$?
26. Снова посмотрите на рис. 3.34. Предположим, что тактовый генератор работает с частотой 40 МГц, а T_{AD} возросло до 16 нс. Можно ли при этом продолжать использовать микросхемы памяти на 40 нс?
27. В табл. 3.4 показано, что T_{ML} должно быть по крайней мере 6 нс. Можете ли вы представить микросхему, у которой этот показатель отрицательный? Другими словами, может ли процессор устанавливать сигнал $\overline{\text{MREQ}}$ до того, как адрес стал стабильным? Объясните почему.
28. Предположим, что передача блока, показанная на рис. 3.38, была произведена на шине с рисунка 3.34. Насколько больше получается пропускная способность при передаче блока по сравнению с отдельными передачами (для длинных блоков)? А теперь предположите, что ширина шины составляет 32 бита вместо 8 битов. Каков будет ваш ответ теперь?
29. Посмотрите на рис. 3.35. Обозначьте время перехода адресных линий как T_{A1} и T_{A2}, время перехода линии $\overline{\text{MREQ}}$ как T_{MREQ1} и T_{MREQ2} и т. д. Напишите все неравенства, подразумеваемые при полном квитировании.
30. Большинство 32-битных шин допускают считывание и запись по 16 битов. Существуют ли какие-нибудь варианты, где именно поместить данные? Аргументируйте.

31. Многие процессоры содержат особый тип цикла шины для подтверждения прерывания. Зачем это нужно?
32. Компьютеру PC/AT, работающему с частотой 10 МГц, требуется 4 цикла, чтобы считать слово. Какую часть пропускной способности шины потребляет процессор?
33. 32-битный процессор с адресными линиями A2–A31 требует, чтобы все ссылки к ячейкам памяти были выровнены. Это значит, что центральный процессор должен обращаться только к словам, состоящим из 4, 8, 12 и т. д. байтов (число байтов кратно 4), и к полусловам, состоящим из четного числа байтов. Байты могут располагаться где угодно. Сколько существует допустимых комбинаций считываний из памяти и сколько требуется выводов, чтобы их выразить? Дайте два ответа.
34. Почему процессор Pentium II не может работать с 32-битной шиной PCI без потери функциональных возможностей? Ведь другие компьютеры с 64-битной шиной могут осуществлять передачи по 32, 16 и даже 8 битов.
35. Предположим, что центральный процессор содержит кэш-память первого и второго уровня со временем доступа 5 нс и 10 нс соответственно. Время доступа к основной памяти составляет 50 нс. Если 20% от всех обращений к памяти приходится на долю кэш-памяти первого уровня, а 60% — на долю кэш-памяти второго уровня, то каково среднее время доступа?
36. Возможно ли, чтобы небольшая встроенная система picoJava II содержала микросхему 8255A?
37. Вычислите пропускную способность шины, необходимую для отображения на мониторе VGA (640×480) цветного фильма (30 кадров/с). Предполагается, что данные должны проходить по шине дважды: один раз от компакт-диска к памяти, а второй раз от памяти к монитору.
38. Как вы думаете, какой сигнал процессора Pentium II запускает линию FRAME# на шине PCI?
39. Какие из сигналов, показанных на рис. 3.49, не являются обязательными для протокола шины?
40. Компьютеру на выполнение каждой команды требуется два цикла шины: один для вызова команды, а второй для вызова данных. Каждый цикл шины занимает 250 нс, а выполнение каждой команды занимает 500 нс (время обработки не принимается в расчет). В компьютере имеется диск. Каждая дорожка этого диска состоит из 16 секторов по 512 байтов. Время обращения диска составляет 8,192 миллисекунд. На сколько процентов снижается скорость работы компьютера во время передачи ПДП (прямой доступ к памяти), если каждая передача ПДП занимает один цикл шины? Рассмотрите два случая: для 8-битных передач и для 16-битных передач по шине.
41. Максимальная полезная нагрузка пакета данных, передаваемого по шине USB, составляет 1023 байта. Если предположить, что устройство может посылать только один пакет данных за кадр, какова максимальная пропускная способность для одного изохронного устройства?

42. Посмотрите на рис. 3.53, *б*. Что получится, если к вентилю НЕ-И, который выбирает микросхему PIO, добавить третью входную линию, связанную с A13?

43. Напишите программу, которая имитирует работу матрицы размером m×n, состоящей из двухвходовых вентилей НЕ-И. Эта схема (она помещается на микросхему) содержит j входных выводов и k выходных выводов. Значения j, k, m и n обрабатываются в процессе компиляции. Программа считывает таблицу монтажных соединений, каждое из соединений определяет вход и выход. Входом может быть либо один из j входных выводов, либо выход какого-нибудь вентиля НЕ-И. Выходом может быть либо один из k выходных выводов, либо вход в какой-нибудь вентиль НЕ-И. Неиспользованные входы принимают значение логической 1. После считывания таблицы соединений программа должна напечатать выход для каждого из 2^j возможных входов. Подобные вентильные матрицы широко используются при нанесении на микросхему схем, разрабатываемых по техническим заданиям заказчика, поскольку большая часть этой работы (имеется в виду нанесение вентильной матрицы на микросхему) не зависит от того, какая это будет схема. Для каждой разработки имеет значение только выбор монтажных соединений.

44. Напишите программу, которая на входе получает два произвольных логических выражения и проверяет, представляют ли они одну и ту же функцию. Входной язык должен включать отдельные буквы (логические переменные), операнды И, ИЛИ и НЕ и скобки. Каждое выражение должно помещаться на одну входную линию. Программа вычисляет таблицы истинности для обеих функций и сравнивает их.

45. Напишите программу, которая получает на входе ряд логических выражений и строит матрицы 24×50 и 50×6, которые нужны для реализации этих выражений в программируемой логической матрице, изображенной на рис. 3.14. Входной язык такой же, как в предыдущем задании. Распечатайте эти матрицы на строчном печатающем устройстве.

Глава 4
Микроархитектурный уровень

Над цифровым логическим уровнем находится микроархитектурный уровень. Его задача — интерпретация команд уровня 2 (уровня архитектуры команд), как показано на рис. 1.2. Строение микроархитектурного уровня зависит от того, каков уровень архитектуры команд, а также от стоимости и предназначения компьютера. В настоящее время уровень архитектуры команд часто содержит простые команды, которые выполняются за один цикл (таковы, в частности, системы RISC). В других системах (например, в системах Pentium II) на этом уровне имеются более сложные команды; выполнение одной такой команды занимает несколько циклов. Чтобы выполнить команду, нужно найти операнды в памяти, считать их и записать полученные результаты обратно в память. Управление уровнем команд со сложными командами отличается от управления уровнем команд с простыми командами, так как в первом случае выполнение одной команды требует определенной последовательности операций.

Пример микроархитектуры

В идеале мы должны были сначала описать общие принципы разработки микроархитектурного уровня. К сожалению, таких общих принципов не существует. Каждая разработка индивидуальна. По этой причине мы просто подробно рассмотрим конкретный пример. В качестве примера мы выбрали подмножество виртуальной машины Java, как мы и обещали в главе 1. Это подмножество содержит только команды с целыми числами, поэтому мы назвали ее **IJVM (Integer JVM; integer — целое число)**. Полную структуру JVM мы рассмотрим в главе 5.

Начнем с описания микроархитектуры, на основе которой мы воплотим IJVM. Система IJVM содержит несколько довольно сложных команд. Подобные архитектуры часто реализуются с помощью микропрограммирования, как уже было сказано в главе 1. Хотя структура IJVM несложная, она послужит отправной точкой в описании основных принципов управления командами и последовательности их выполнения.

Наша микроархитектура содержит микропрограмму (в ПЗУ), которая должна вызывать, декодировать и выполнять команды IJVM. Мы не можем использовать для этой микропрограммы интерпретатор JVM, разработанный компанией Sun,

поскольку нам нужна крошечная микропрограмма, которая запускает отдельные вентили аппаратного обеспечения. Интерпретатор JVM компании Sun был написан на языке C, чтобы обеспечить мобильность программного обеспечения. Этот интерпретатор не может управлять аппаратным обеспечением на таком детализированном уровне, который нам нужен. Поскольку реальное аппаратное обеспечение состоит только из компонентов, описанных в главе 3, то теоретически после изучения этой главы читатель сможет пойти в магазин, купить огромное количество транзисторов и сконструировать машину IJVM. Тому, кто успешно выполнит эту задачу, будет предоставлен дополнительный кредит (а также полное психиатрическое обследование).

Разработку данной микроархитектуры удобно считать проблемой программирования, при этом каждая команда уровня архитектуры команд — функция, вызываемая основной программой. В данном случае основная программа довольно проста. Она представляет собой бесконечный цикл. Сначала программа определяет, какую функцию нужно выполнить, затем вызывает эту функцию, а затем начинает все снова.

Микропрограмма содержит набор переменных, к которым имеют доступ все функции. Этот набор переменных называется **состоянием** компьютера. Каждая функция изменяет по крайней мере несколько переменных, формируя при этом состояние. Например, счетчик команд — это часть состояния. Он указывает местонахождение функции (то есть команды уровня архитектуры команд), которую нужно выполнить следующей. Во время выполнения каждой команды счетчик команд указывает на следующую команду.

Команды IJVM очень короткие. Каждая команда состоит из нескольких полей, обычно одного или двух, каждое из которых выполняет определенную функцию. Первое поле является **кодом операции**. Этот код определяет тип команды и сообщает, что это, например, команда сложения или команда ветвления, или еще какая-нибудь команда. Многие команды содержат дополнительное поле, которое определяет тип операнда. Например, команды, которые имеют доступ к локальным переменным, должны иметь специальное поле, чтобы определить, какая это переменная.

Такая модель выполнения команды, называемая иногда **циклом выборка-исполнение**, полезна для теории, а также может служить основой воплощения уровня архитектуры команд со сложными командами (например, IJVM). Ниже мы опишем, как работает эта модель, что собой представляет микроархитектура и как ею управляют микрокоманды, каждая из которых занимает тракт данных на один цикл. Полный список команд формирует микропрограмму, которая будет рассмотрена очень подробно.

Тракт данных

Тракт данных — это часть центрального процессора, состоящая из АЛУ (арифметико-логического устройства) и его входов и выходов. Тракт данных нашей микроархитектуры показан на рис. 4.1. Хотя этот тракт данных и был оптимизирован для интерпретации программ IJVM, он схож с трактами данных большинства компьютеров. Он содержит ряд 32-разрядных регистров, которым мы приписали символические названия (например, PC, SP, MDR). Хотя некоторые из этих названий

нам знакомы, важно понимать, что эти регистры доступны только на микроархитектурном уровне (для микропрограммы). Им даны такие названия, поскольку они обычно содержат значения, соответствующие переменным с аналогичными названиями на уровне архитектуры команд. Содержание большинства регистров передается на шину B. Выходной сигнал АЛУ запускает схему сдвига, а затем шину C. Значение из шины C может записываться в один или несколько регистров одновременно. Шину A мы введем позже, а пока представим, что ее нет.

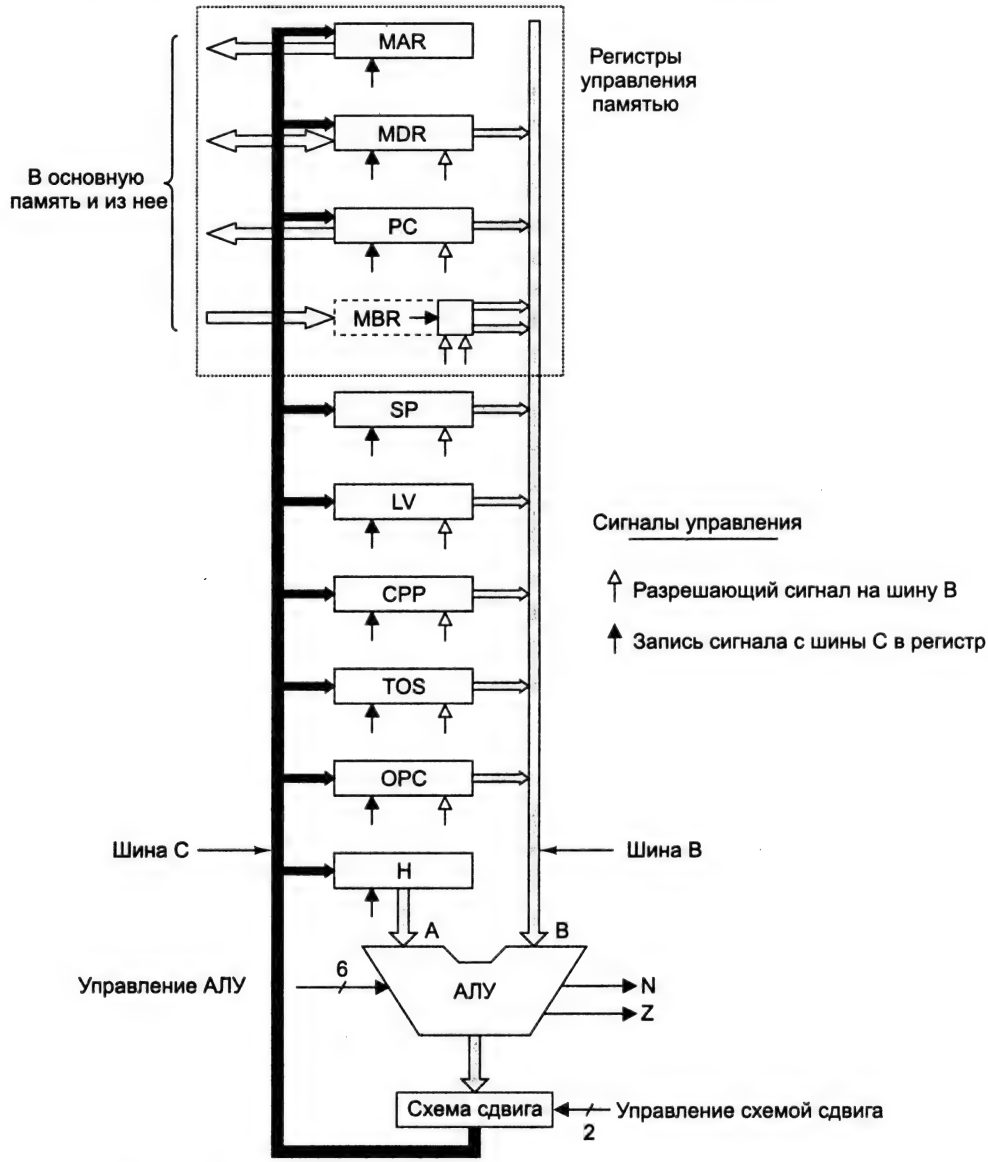


Рис. 4.1. Тракт данных микроархитектуры, рассматриваемой в этой главе

Данное АЛУ идентично тому, которое изображено на рис. 3.18 и 3.19. Его функционирование зависит от линий управления. На рис. 4.1 перечеркнутая стрелочка с цифрой 6 сверху указывает на наличие шести линий управления АЛУ. Из них F_0 и F_1 служат для определения операции, ENA и ENB — для разрешения входных сигналов A и B соответственно, INVA — для инверсии левого входа и INC — для прибавления единицы к результату. Однако не все 64 комбинации значений на линиях управления могут быть полезны.

Некоторые комбинации показаны в табл. 4.1. Не все из этих функций нужны для IJVM, но многие из них могут пригодиться для полной JVM. В большинстве случаев существует несколько возможностей для достижения одного и того же результата. В данной таблице знак «+» означает арифметический плюс, а знак «–» — арифметический минус, поэтому –A означает дополнение A.

Таблица 4.1. Некоторые комбинации сигналов АЛУ и соответствующие им функции

F_0	F_1	ENA	ENB	INVA	INC	Функция
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\bar{A}$
1	0	1	1	0	0	$\bar{B}$
1	1	1	1	0	0	A+B
1	1	1	1	0	1	A+B+1
1	1	1	0	0	1	A+1
1	1	0	1	0	1	B+1
1	1	1	1	1	1	B–A
1	1	0	1	1	0	B–1
1	1	1	0	1	1	–A
0	0	1	1	0	0	A И B
0	1	1	1	0	0	A ИЛИ B
0	1	0	0	0	0	0
0	1	0	0	0	1	1
0	1	0	0	1	0	–1

АЛУ, изображенное на рис. 4.1, содержит два входа для данных: левый вход (A) и правый вход (B). С левым входом связан регистр временного хранения H. С правым входом связана шина B, в которую могут поступать значения из одного из девяти источников, что показано с помощью девяти серых стрелок, примыкающих к шине. Существует и другая разработка АЛУ с двумя полноразрядными шинами, и мы рассмотрим ее чуть позже в этой главе.

В регистр H может поступать функция АЛУ, которая проходит через правый вход (из шины B) к выходу АЛУ. Одна из таких функций — сложение входных сигналов АЛУ, только при этом сигнал ENA отрицается, и левый вход получает значение 0. Если к значению шины B прибавить 0, это значение не изменится. Затем результат проходит через схему сдвига (также без изменений) и сохраняется в регистре H.

Существует еще две линии управления, которые используются независимо от остальных. Они служат для управления выходом АЛУ. Линия SLL8 (Shift Left Logical — логический сдвиг влево) сдвигает число влево на 1 байт, заполняя 8 самых младших двоичных разрядов нулями; линия SRA1 (Shift Right Arithmetic — арифметический сдвиг вправо) число вправо на 1 бит, оставляя самый старший двоичный разряд без изменений.

Можно считать и записать один и тот же регистр за один цикл. Для этого, например, нужно поместить значение SP на шину B, закрыть левый вход АЛУ, установить сигнал INC и сохранить полученный результат в регистре SP, увеличив таким образом его значение на 1 (см. восьмую строку табл. 4.1). Если один и тот же регистр может считываться и записываться за один цикл, то как при этом предотвратить появление ненужных данных? Дело в том, что процессы чтения и записи проходят в разных частях цикла. Когда в качестве правого входа АЛУ выбирается один из регистров, его значение помещается на шину B в начале цикла и хранится там на протяжении всего цикла. Затем АЛУ выполняет свою работу и производит результат, который через схему сдвига поступает на шину C. Незадолго до конца цикла, когда значения выходных сигналов АЛУ и схемы сдвига стабилизировались, содержание шины C передается в один или несколько регистров. Одним из этих регистров вполне может быть тот, от которого поступил сигнал на шину B. Точная синхронизация тракта данных делает возможным считывание и запись одного и того же регистра за один цикл. Об этом речь пойдет ниже.

Синхронизация тракта данных

Как происходит синхронизация этих действий, показано на рис. 4.2. Здесь в начале каждого цикла генерируется короткий импульс. Он может выдаваться задающим генератором, как показано на рис. 3.20, *в*. На заднем фронте импульса устанавливаются биты, которые будут запускать все вентили. Этот процесс занимает определенный отрезок времени Δw. Затем выбирается регистр, и его значение передается на шину B. На это требуется время Δx. Затем АЛУ и схема сдвига начинают оперировать поступившими к ним данными. После промежутка Δy выходные сигналы АЛУ и схемы сдвига стабилизируются. В течение следующего отрезка Δz результаты проходят по шине C к регистрам, куда они загружаются на нарастающем фронте следующего импульса. Загрузка должна запускаться фронтом сигнала и осуществляться мгновенно, так что даже в случае изменений каких-либо входных регистров изменения в шине C будут происходить только после полной загрузки регистров. На нарастающем фронте импульса регистр, запускающий шину B, приостанавливает свою работу и ждет следующего цикла. На рис. 4.2 упомянуты регистры MPC и MIR, а также память. Их предназначение мы обсудим чуть позже.

Важно осознавать, что хотя в тракте данных нет никаких запоминающих элементов, для прохождения сигнала по нему требуется определенное время. Изменение значения на шине B вызывает изменения на шине C не сразу, а только через некоторое время (это объясняется задержками на каждом шаге). Следовательно, даже если один из входных регистров изменяется, новое значение будет сохранено

в регистре задолго до того, как старое (и уже неправильное) значение этого регистра, помещенное на шину B, сможет достичь АЛУ.

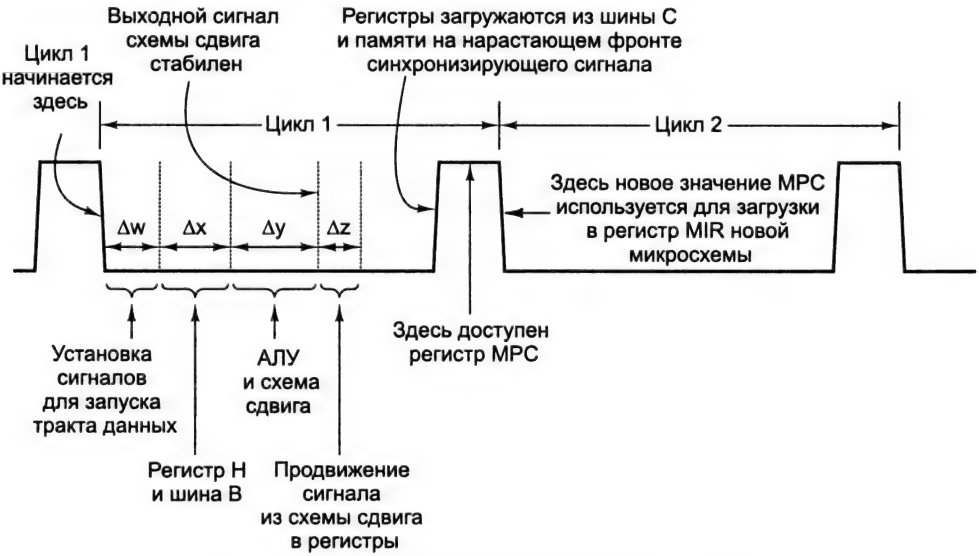


Рис. 4.2. Временная диаграмма цикла тракта данных

Для такой разработки требуется жесткая синхронизация и довольно длинный цикл; должно быть известно минимальное время прохождения сигнала через АЛУ; регистры должны загружаться из шины C очень быстро. Если подойти к этому вопросу с особым вниманием и осторожностью, можно сделать так, чтобы тракт данных функционировал правильно.

Цикл тракта данных можно разбить на подциклы. Начало подцикла 1 запускается задним фронтом синхронизирующего сигнала. Ниже показано, что происходит во время каждого из подциклов. В скобках приводится длина подцикла.

1. Устанавливаются сигналы управления (Δw).
2. Значения регистров загружаются на шину B (Δx).
3. Происходит работа АЛУ и схемы сдвига (Δy).
4. Результаты проходят по шине C обратно к регистрам (Δz).

На нарастающем фронте следующего цикла результаты сохраняются в регистрах.

Никаких внешних сигналов, указывающих на начало и конец подцикла и сообщающих АЛУ, когда нужно начинать работу и когда нужно передавать результаты на шину C, нет. В действительности АЛУ и схема сдвига работают постоянно. Однако их входные сигналы недействительны в течение периода Δw+Δx. Точно так же их выходные сигналы недействительны в течение периода Δw+ Δx+Δy. Единственными внешними сигналами, управляющими трактом данных, являются задний фронт синхронизирующего сигнала, с которого начинается цикл тракта данных, и нарастающий фронт синхронизирующего сигнала, который загружает регистры из шины C. Границы подциклов определяются только временем прохождения сигнала, поэтому разработчики тракта данных должны все очень четко рассчитать.

Работа памяти

Наша машина может взаимодействовать с памятью двумя способами: через порт с пословной адресацией (32-битный) и через порт с байтовой адресацией (8-битный). Порт с пословной адресацией управляется двумя регистрами: **MAR (Memory Address Register — регистр адреса ячейки памяти)** и **MDR (Memory Data Register — информационный регистр памяти)**, которые показаны на рис. 4.1. Порт с байтовой адресацией управляется регистром PC, который записывает 1 байт в 8 младших разрядов регистра **MBR (Memory Buffer Register — буферный регистр памяти)**. Этот порт может считывать данные из памяти, но не может их записывать в память.

Каждый из этих регистров, а также все остальные регистры, изображенные на рис. 4.1, запускаются одним из **сигналов управления**. Белая стрелка под регистром указывает на сигнал управления, который разрешает передавать выходной сигнал регистра на шину B. Регистр MAR не связан с шиной B, поэтому у него нет сигнала разрешения. У регистра H этого сигнала тоже нет, так как он является единственным возможным левым входом АЛУ и поэтому всегда разрешен.

Черная стрелка под регистром указывает на сигнал управления, который записывает (то есть загружает) регистр из шины C. Поскольку регистр MBR не может загружаться из шины C, у него нет сигнала записи (но зато есть два сигнала разрешения, о которых речь пойдет ниже). Чтобы инициировать процесс считывания из памяти или записи в память, нужно загрузить соответствующие регистры памяти, а затем передать памяти сигнал чтения или записи (он не показан на рис. 4.1).

Регистр MAR содержит адреса слов, таким образом, значения 0, 1, 2 и т. д. указывают на последовательные слова. Регистр PC содержит адреса байтов, таким образом, значения 0, 1, 2 и т. д. указывают на последовательные байты. Если значение 2 поместить в регистр PC и начать процесс чтения, то из памяти считается байт 2, который затем будет записан в 8 младших разрядов регистра MBR. Если значение 2 поместить в регистр MAR и начать процесс чтения, то из памяти считаются байты 8–11 (то есть слово 2), которые затем будут записаны в регистр MDR.

Для чего потребовалось два регистра с разной адресацией? Дело в том, что регистры MAR и PC будут использоваться для обращения к двум разным частям памяти, а зачем это нужно, станет ясно чуть позже. А пока достаточно сказать, что регистры MAR и MDR используются для чтения и записи слов данных на уровне архитектуры команд, а регистры PC и MBR — для считывания программы уровня архитектуры команд, которая состоит из потока байтов. Во всех остальных регистрах, содержащих адреса, применяется принцип пословной адресации, как и в MAR.

В действительности существует только одна память: с байтовой адресацией. Как же регистр MAR обращается к словам, если память состоит из байтов? Когда значение регистра MAR помещается на адресную шину, 32 бита этого значения не попадают точно на 32 адресные линии (с 0 по 31). Вместо этого бит 0 соединяется с адресной линией 2, бит 1 — с адресной линией 3 и т. д. Два старших бита не учитываются, поскольку они нужны только для адресов свыше 2^{32}, а такие адреса недопустимы в нашей машине на 4 Гбайт. Когда значение MAR равно 1, на шину помещается адрес 4; когда значение MAR равно 2, на шину помещается адрес 8 и т. д. Распределение битов регистра MAR по адресным линиям показано на рис. 4.3.



Рис. 4.3. Распределение битов регистра MAR в адресной шине

Как уже было сказано выше, данные, считанные из памяти через 8-битный порт, сохраняются в 8-битном регистре MBR. Этот регистр может быть скопирован на шину B двумя способами: со знаком и без знака. Когда требуется значение без знака, 32-битное слово, помещаемое на шину B, содержит значение MBR в младших 8 битах и нули в остальных 24 битах. Значения без знака нужны для индексирования таблиц или для получения целого 16-битного числа из двух последовательных байтов (без знака) в потоке команд.

Другой способ превращения 8-битного регистра MBR в 32-битное слово — рассматривать его как значение со знаком между −128 и +127 и использовать это значение для порождения 32-битного слова с тем же самым численным значением. Это преобразование делается путем дублирования знакового бита (самого левого бита) регистра MBR в верхние 24 битовые позиции шины B. Такой процесс называется **расширением по знаку** или **знаковым расширением**. Если выбран данный параметр, то либо все старшие 24 бита примут значение 0, либо все они примут значение 1, в зависимости от того, каков самый левый бит регистра MBR: 0 или 1.

В какое именно 32-битное значение (со знаком или без знака) превратится 8-битное значение регистра MBR, определяется тем, какой из двух сигналов управления (две белые стрелки под регистром MBR на рис. 4.1) установлен. Прямоугольник, обозначенный на рисунке пунктиром, показывает способность 8-битного регистра MBR действовать в качестве источника 32-битных слов для шины B.

Микрокоманды

Для управления трактом данных, изображенным на рис. 4.1, нам нужно 29 сигналов. Их можно разделить на пять функциональных групп:

- 9 сигналов для записи данных из шины C в регистры.
- 9 сигналов для разрешения передачи регистров на шину B и в АЛУ.
- 8 сигналов для управления АЛУ и схемой сдвига.
- 2 сигнала, которые указывают, что нужно осуществить чтение или запись через регистры MAR/MDR (на рисунке они не показаны).
- 1 сигнал, который указывает, что нужно осуществить вызов из памяти через регистры PC/MBR (на рисунке также не показан).

Значения этих 29 сигналов управления определяют операции для одного цикла тракта данных. Цикл состоит из передачи значений регистров на шину B, прохождения этих сигналов через АЛУ и схему сдвига, передачи полученных результатов на шину C и записи их в нужный регистр (регистры). Кроме того, если установлен сигнал считывания данных, то в конце цикла после загрузки регистра MAR начинается работа памяти. Данные из памяти помещаются в MBR или MDR в конце *следующего* цикла, а использоваться эти данные могут в цикле, который идет *после* него. Другими словами, если считывание из памяти через любой из портов начинается в конце цикла k, то полученные данные еще не могут использоваться в цикле k+1 (только в цикле k+2 и позже).

Этот процесс объясняется на рис. 4.2. Сигналы управления памятью выдаются только после загрузки регистров MAR и PC, которая происходит на нарастающем фронте синхронизирующего сигнала незадолго до конца цикла 1. Мы предположим, что память помещает результаты на шину памяти в течение одного цикла, поэтому регистры MBR и (или) MDR могут загружаться на следующем нарастающем фронте вместе с другими регистрами.

Другими словами, мы загружаем регистр MAR в конце цикла тракта данных и запускаем память сразу после этого. Следовательно, мы не можем ожидать, что результаты считывания будут в регистре MDR в начале следующего цикла, особенно если длительность импульса небольшая. Этого времени будет недостаточно. Поэтому между началом считывания из памяти и использованием этого результата должен помещаться один цикл. Однако во время этого цикла может выполняться не только передача слова из памяти, но и другие операции.

Предположение о том, что работа памяти занимает один цикл, эквивалентно предположению, что количество успешных обращений в кэш-память составляет 100%. Подобное предположение никогда не может быть истинным, но мы не будем здесь рассказывать о циклах памяти переменной длины, поскольку это не входит в задачи данной книги.

Так как регистры MBR и MDR загружаются на нарастающем фронте синхронизирующего сигнала вместе с другими регистрами, они могут считывать во время циклов, в течение которых осуществляется передача нового слова из памяти. Они возвращают старые значения, поскольку прошло еще недостаточно времени для того, чтобы поменять их на новые. Здесь нет никакой двусмысленности: до тех пор пока новые значения не загрузятся в регистры MBR и MDR на нарастающем фронте сигнала, предыдущие значения находятся там и могут использоваться. Отметим, что считывания могут проходить одно за другим, то есть в двух последовательных циклах (поскольку сам процесс считывания занимает только один цикл). Кроме того, обе памяти могут действовать в одно и то же время. Однако попытка чтения и записи одного и того же байта одновременно приводит к неопределенным результатам.

Выходной сигнал шины C можно записывать сразу в несколько регистров, однако нежелательно передавать значения более одного регистра на шину B. Немного расширив схемотехнику, мы можем сократить количество битов, необходимых для выбора одного из возможных источников для запуска шины B. Существует только

9 входных регистров, которые могут запустить шину B (регистры MBR со знаком и без знака учитываются отдельно). Следовательно, мы можем закодировать информацию для шины B в 4 бита и использовать декодер для порождения 16 сигналов управления, 7 из которых не нужны. У разработчиков коммерческих моделей, возможно, было бы большое желание избавиться от одного из регистров, чтобы обойтись 3 битами. Однако мы как ученые предпочитаем иметь один лишний бит, но при этом получить более ясную и простую разработку.

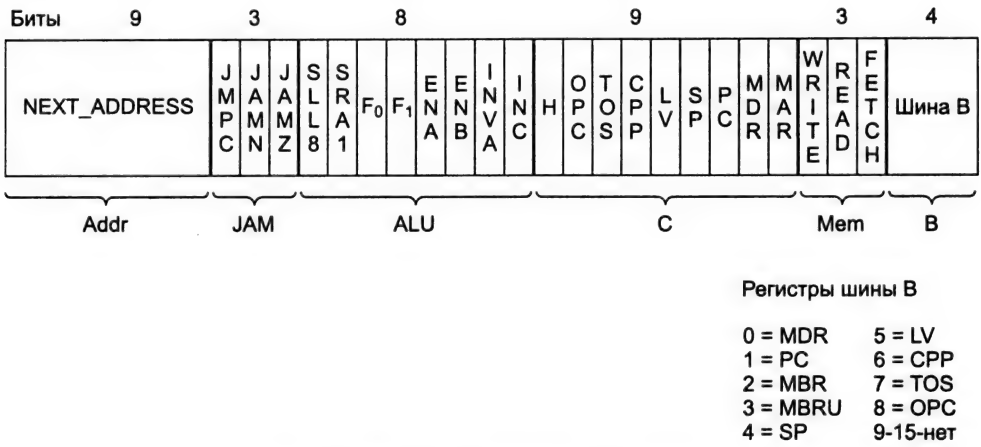


Рис. 4.4. Формат микрокоманды для Mic-1

Теперь мы можем управлять трактом данных с помощью 9+4+8+2+1=24 сигналов, следовательно, нам требуется 24 бита. Однако эти 24 бита управляют трактом данных только в течение одного цикла. Задача управления — определить, что нужно делать в следующем цикле. Чтобы включить это в разработку контроллера, мы создадим формат для описания операций, которые нужно выполнить, используя 24 бита управления и два дополнительных поля: поле NEXT_ADDRESS (следующий адрес) и поле JAM. Содержание каждого из этих полей мы обсудим позже. На рис. 4.4 изображен один из возможных форматов. Он разделен на следующие 6 групп, содержащие 36 сигналов:

- Addr — содержит адрес следующей потенциальной микрокоманды.
- JAM — определяет, как выбирается следующая микрокоманда.
- ALU — функции АЛУ и схемы сдвига.
- C — выбирает, какие регистры записываются из шины C.
- Mem — функции памяти.
- B — выбирает источник для шины B (как он кодируется, было показано выше).

Порядок групп в принципе произволен, хотя мы долго и тщательно его подбирали, чтобы избежать пересечений на рис. 4.5. Подобные пересечения на диаграммах часто соответствуют пересечениям проводов на микросхемах. Они сильно затрудняют разработку и их лучше сводить к минимуму.

Управление микрокомандами: Mic-1

До сих пор мы рассказывали о том, как происходит управление трактом данных, но мы еще не касались того, каким образом решается, какой именно сигнал управления и на каком цикле должен запускаться. Для этого существует **контроллер последовательности**, который отвечает за последовательность операций, необходимых для выполнения одной команды.

Контроллер последовательности в каждом цикле должен выдавать следующую информацию:

1. Состояние каждого сигнала управления в системе.
2. Адрес микрокоманды, которая будет выполняться следующей.

Рисунок 4.5 представляет собой подробную диаграмму полной микроархитектуры нашей машины, которую мы назовем **Mic-1**. На первый взгляд она может показаться внушительной, но тем не менее ее нужно подробно изучить. Если вы разберетесь во всех прямоугольниках и линиях, изображенных на этом рисунке, вам легче будет понять структуру микроархитектурного уровня. Диаграмма состоит из двух частей: тракта данных (слева), который мы уже подробно обсудили, и блока управления (справа), который мы рассмотрим сейчас.

Самой большой и самой важной частью блока управления является **управляющая память**. Удобно рассматривать ее как память, в которой хранится полная микропрограмма, хотя иногда она реализуется в виде набора логических вентилей. Мы будем называть ее управляющей памятью, чтобы не путать с основной памятью, доступ к которой осуществляется через регистры MBR и MDR. Функционально управляющая память представляет собой память, которая содержит микрокоманды вместо обычных команд. В нашем примере она содержит 512 слов, каждое из которых состоит из одной 32-битной микрокоманды с форматом, изображенным на рис. 4.4. В действительности не все эти слова нужны, но по ряду причин нам требуются адреса для 512 отдельных слов.

Управляющая память отличается от основной памяти тем, что команды, хранящиеся в основной памяти, выполняются в порядке адресов (за исключением ветвлений), а микрокоманды — нет. Увеличение счетчика команд в листинге 2.1 означает, что команда, которая будет выполняться после текущей, — это команда, которая идет вслед за текущей в памяти. Микропрограммы должны обладать большей гибкостью (поскольку последовательности микрокоманд обычно короткие), поэтому они не обладают этим свойством. Вместо этого каждая микрокоманда сама указывает на следующую микрокоманду.

Поскольку управляющая память функционально представляет собой ПЗУ, ей нужен собственный адресный регистр и собственный регистр данных. Ей не требуются сигналы чтения и записи, поскольку здесь постоянно происходит процесс считывания. Мы назовем адресный регистр управляющей памяти **MPC (MicroProgram Counter — микропрограммный счетчик)**. Название не очень подходящее, поскольку микропрограммы не упорядочены явным образом и понятие счетчика тут неуместно, но мы не можем пойти против традиций. Регистр данных мы назовем **MIR (MicroInstruction Register — регистр микрокоманд)**. Он содержит текущую микрокоманду, биты которой запускают сигналы управления, влияющие на работу тракта данных.

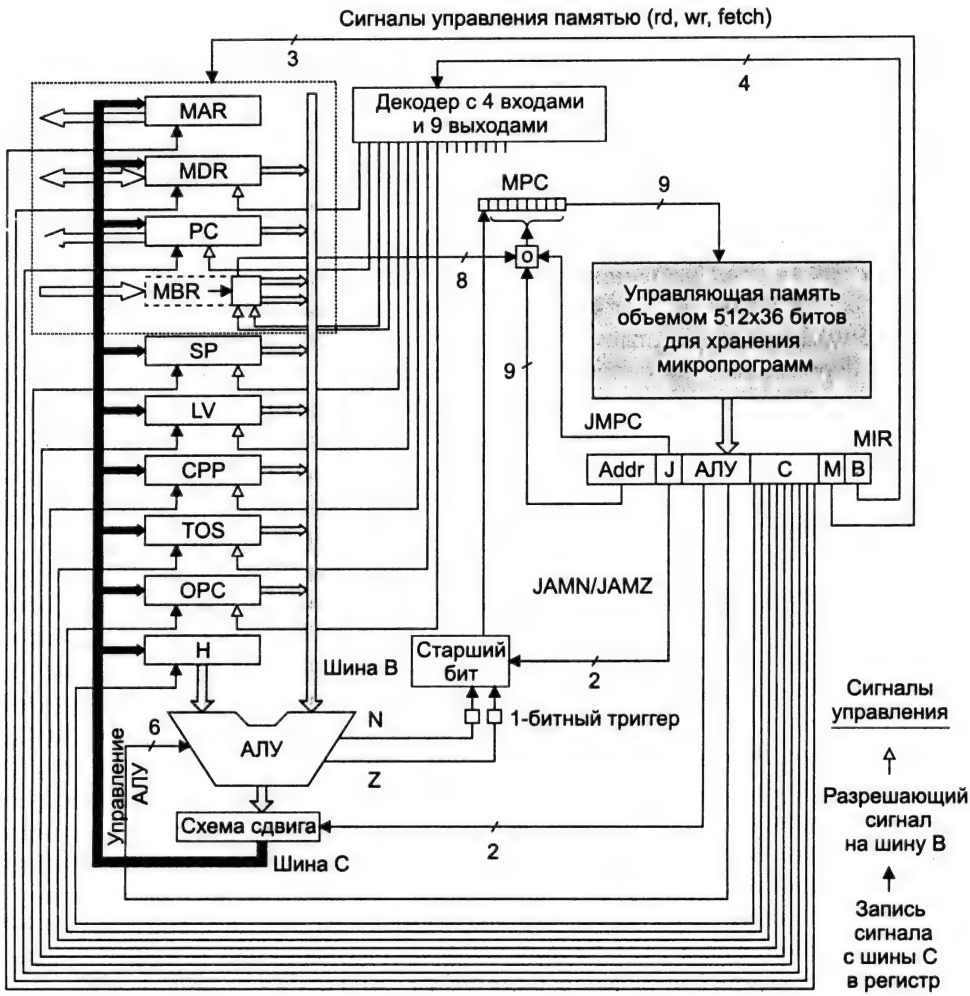


Рис. 4.5. Полная диаграмма микроархитектуры Mic-1

Регистр MIR, изображенный на рис. 4.5, содержит те же шесть групп сигналов, которые показаны на рис. 4.4. Группы Addr и J (то же, что JAM) контролируют выбор следующей микрокоманды. Мы обсудим их чуть позже. Группа ALU содержит 8 битов, которые выбирают функцию АЛУ и запускают схему сдвига. Биты C загружают отдельные регистры из шины C. Сигналы M управляют работой памяти.

Наконец, последние 4 бита запускают декодер, который определяет, значение какого регистра будет передано на шину B. В данном случае мы выбрали декодер, который содержит 4 входа и 16 выходов, хотя имеется всего 9 разных регистров. В более проработанной модели мог бы использоваться декодер, имеющий 4 входа и 9 выходов. Мы используем стандартную схему, чтобы не разрабатывать свою собственную. Использовать стандартную схему гораздо проще, и кроме того, вы сможете избежать ошибок. Ваша собственная микросхема займет меньше места,

но на ее разработку потребуется довольно длительное время, к тому же вы можете построить ее неправильно.

Схема, изображенная на рис. 4.5, работает следующим образом. В начале каждого цикла (задний фронт синхронизирующего сигнала на рис. 4.2) в регистр MIR загружается слово из управляющей памяти, которая на рисунке отмечена буквами MPC. Загрузка регистра MIR занимает период Δw, то есть первый подцикл (см. рис. 4.2).

Когда микрокоманда попадает в MIR, в тракт данных поступают различные сигналы. Значение определенного регистра помещается на шину B, а АЛУ узнает, какую операцию нужно выполнять. Все это происходит во время второго подцикла. После периода $\Delta w+\Delta x$ входные сигналы АЛУ стабилизируются.

После периода Δy стабилизируются сигналы АЛУ N и Z и выходной сигнал схемы сдвига. Затем значения N и Z сохраняются в двух 1-битных триггерах. Эти биты, как и все регистры, которые загружаются из шины C и из памяти, сохраняются на нарастающем фронте синхронизирующего сигнала, ближе к концу цикла тракта данных. Выходной сигнал АЛУ не сохраняется, а просто передается в схему сдвига. Работа АЛУ и схемы сдвига происходит во время подцикла 3.

После следующего интервала, Δz, выходной сигнал схемы сдвига, пройдя через шину C, достигает регистров. Регистры загружаются в конце цикла на нарастающем фронте синхронизирующего сигнала (см. рис. 4.2). Во время подцикла 4 происходит загрузка регистров и триггеров N и Z. Он завершается сразу после нарастающего фронта, когда все значения сохранены, результаты предыдущих операций памяти доступны и регистр MPC загружен. Этот процесс продолжается снова и снова, пока вы не устанете и не выключите компьютер.

Микропрограмме приходится не только управлять трактом данных, но и определять, какая микрокоманда должна быть выполнена следующей, поскольку они не упорядочены в управляющей памяти. Вычисление адреса следующей микрокоманды начинается после загрузки регистра MIR. Сначала в регистр MPC копируется 9-битное поле NEXT_ADDRESS (следующий адрес). Пока происходит копирование, проверяется поле JAM. Если оно содержит значение 000, то ничего больше делать не нужно; когда копирование поля NEXT_ADDRESS завершится, регистр MPC укажет на следующую микрокоманду.

Если один или несколько бит в поле JAM равны 1, то требуются еще некоторые действия. Если бит JAMN равен 1, то триггер N соединяется через схему ИЛИ со старшим битом регистра MPC. Если бит JAMZ равен 1, то триггер Z соединяется через схему ИЛИ со старшим битом регистра MPC. Если оба бита равны 1, они оба соединяются через схему ИЛИ с тем же битом. А теперь объясним, зачем нужны триггеры N и Z. Дело в том, что после нарастающего фронта сигнала (и вплоть до заднего фронта) шина B больше не запускается, поэтому выходные сигналы АЛУ уже не могут считаться правильными. Сохранение флагов состояния АЛУ в регистрах N и Z делает правильные значения стабильными и доступными для вычисления регистра MPC, независимо от того, что происходит вокруг АЛУ.

На рис. 4.5 схема, которая выполняет это вычисление, называется «старший бит». Она вычисляет следующую булеву функцию:

F = ((JAMZ И Z) ИЛИ (JAMN И N)) ИЛИ NEXT_ADDRESS[8]

Отметим, что в любом случае регистр MPC может принять только одно из двух возможных значений:

1. Значение NEXT_ADDRESS.
2. Значение NEXT_ADDRESS со старшим битом, соединенным с логической единицей операцией ИЛИ.

Других значений не существует. Если старший бит значения NEXT_ADDRESS уже равен 1, нет смысла использовать JAMN или JAMZ.

Отметим, что если все биты JAM равны 0, то адрес следующей команды — просто 9-битный номер в поле NEXT_ADDRESS. Если JAMN или JAMZ равны 1, то существует два потенциально возможных адреса следующей микрокоманды: NEXT_ADDRESS и NEXT_ADDRESS, соединенный операцией ИЛИ с 0x100 (предполагается, что NEXT_ADDRESS≤0xFF). (Отметим, что 0x указывает, что число, следующее за ним, дается в шестнадцатеричной системе счисления). Это проиллюстрировано рис. 4.6. Текущая микрокоманда с адресом 0x75 содержит поле NEXT_ADDRESS=0x92, причем бит JAMZ установлен на 1. Следовательно, следующий адрес микрокоманды зависит от значения бита Z, сохраненного при предыдущей операции АЛУ. Если бит Z равен 0, то следующая микрокоманда имеет адрес 0x92. Если бит Z равен 1, то следующая микрокоманда имеет адрес 0x192.

Третий бит в поле JAM — JMPC. Если он установлен, то 8 битов регистра MBR поразрядно связываются операцией ИЛИ с 8 младшими битами поля NEXT_ADDRESS из текущей микрокоманды. Результат отправляется в регистр MPC. На рис. 4.5 значком «ИЛИ» обозначена схема, которая выполняет операцию ИЛИ над MBR и NEXT_ADDRESS, если бит JMPC равен 1, и просто отправляет NEXT_ADDRESS в регистр MPC, если бит JMPC равен 0. Если JMPC равен 1, то младшие 8 битов поля NEXT_ADDRESS равны 0. Старший бит может быть 0 или 1, поэтому значение поля NEXT_ADDRESS обычно 0x000 или 0x100. Почему иногда используется 0x000, а иногда — 0x100, мы обсудим позже.

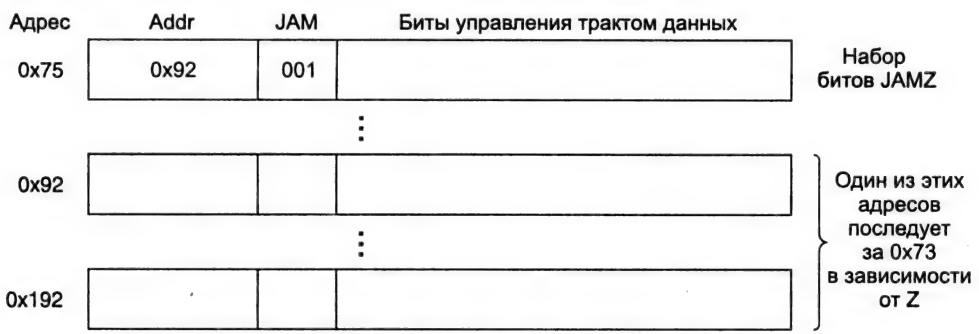


Рис. 4.6. Микрокоманда с битом JAMZ, равным 1, указывает на две потенциальные последующие микрокоманды

Возможность выполнять операцию ИЛИ над MBR и NEXT_ADDRESS и сохранять результат в регистре MPC позволяет реализовывать межуровневые переходы. Отметим, что по битам, находящимся в регистре MBR, можно определить любой адрес из 256 возможных. Регистр MBR содержит код операции, поэтому использование JMPC приведет к единственно возможному выбору следующей

микрокоманды. Этот метод позволяет осуществлять быстрый переход у функции, соответствующей вызванному коду операции.

Для того чтобы продолжить чтение этой главы, очень важно понимать принципы синхронизации машины, поэтому повторим их еще раз. Синхронизирующий сигнал делится на подциклы, хотя внешние изменения этого сигнала происходят только на заднем фронте, с которого начинается цикл, и на нарастающем фронте, который загружает регистры и триггеры N и Z. Посмотрите еще раз на рис. 4.2.

Во время подцикла 1, который инициируется задним фронтом сигнала, адрес, находящийся в данный момент в регистре MPC, загружается в регистр MIR. Во время подцикла 2 регистр MIR выдает сигналы и в шину B загружается выбранный регистр. Во время подцикла 3 происходит работа АЛУ и схемы сдвига. Во время подцикла 4 стабилизируются значения шины C, шин памяти и АЛУ. На нарастающем фронте сигнала загружаются регистры из шины C, загружаются триггеры N и Z, а регистры MBR и MDR получают результаты работы памяти, начавшейся в конце предыдущего цикла (если эти результаты вообще имеются). Как только регистр MBR получает свое значение, загружается регистр MPC. Это происходит где-то в середине отрезка между нарастающим и задним фронтами, но уже после загрузки MBR/MDR. Он может загружаться уровнем сигнала (но не фронтом сигнала) либо загружаться через фиксированный отрезок времени после нарастающего фронта. Все это означает, что регистр MPC не получает своего значения до тех пор, пока не будут готовы регистры MBR, N и Z, от которых он зависит. На заднем фронте сигнала, когда начинается новый цикл, регистр MPC может обращаться к памяти.

Отметим, что каждый цикл является самодостаточным. В каждом цикле определяется, значение какого регистра должно поступать на шину B, что должны делать АЛУ и схема сдвига, куда нужно сохранить значение шины C, и, наконец, каким должно быть следующее значение регистра MPC.

Следует сделать еще одно замечание по поводу рис. 4.5. До сих пор мы считали MPC регистром, который состоит из 9 битов и загружается на высоком уровне сигнала. В действительности этот регистр вообще не нужен. Все его входные сигналы можно непосредственно связать с управляющей памятью. Поскольку они имеются в управляющей памяти на заднем фронте синхронизирующего сигнала, когда выбирается и считывается регистр MIR, этого достаточно. Их не нужно хранить в регистре MPC. По этой причине MPC может быть реализован в виде **виртуального регистра**, который представляет собой просто место скопления сигналов и похож скорее на коммутационное поле, чем на настоящий регистр. Если MPC сделать виртуальным регистром, то процедура синхронизации сильно упрощается: теперь события происходят только на нарастающем фронте и заднем фронте сигнала. Но если вам проще считать MPC реальным регистром, то такой подход тоже вполне допустим.

Пример архитектуры команд: IJVM

Чтобы продолжить описание нашего примера, введем уровень набора команд, которые должна интерпретировать микропрограмма машины IJVM (см. рис. 4.5). Для удобства уровень архитектуры команд мы иногда будем называть **макроархитек-**

турой, чтобы противопоставить его микроархитектуре. Однако перед тем как приступать к описанию IJVM, мы немного отвлечемся.

Стек

Во всех языках программирования есть понятие процедур с локальными переменными. Эти переменные доступны во время выполнения процедуры, но перестают быть доступными после окончания процедуры. Возникает вопрос: где должны храниться такие переменные?

К сожалению, предоставить каждой переменной абсолютный адрес в памяти невозможно. Проблема заключается в том, что процедура может вызывать себя сама. Мы рассмотрим такие рекурсивные процедуры в главе 5. А пока достаточно сказать, что если процедура вызывается дважды, то хранить ее переменные под конкретными адресами в памяти нельзя, поскольку второй вызов нарушит результаты первого.

Вместо этого используется другая стратегия. Для переменных резервируется особая область памяти, которая называется **стеком**, но отдельные переменные не получают в нем абсолютных адресов. Какой-либо регистр, скажем, LV, указывает на базовый адрес локальных переменных для текущей процедуры. Рассмотрим рис. 4.7, *а*. В данном случае вызывается процедура A с локальными переменными *a1*, *a2* и *a3*, и для этих переменных резервируется участок памяти, начинающийся с адреса, который указывается регистром LV. Другой регистр, SP, указывает на старшее слово локальных переменных процедуры A. Если значение регистра LV равно 100, а слова состоят из 4 байтов, то значение SP будет 108. Для обращения к переменной нужно вычислить ее смещение от адреса LV. Структура данных между LV и SP (включая оба указанных слова) называется **фреймом локальных переменных**.

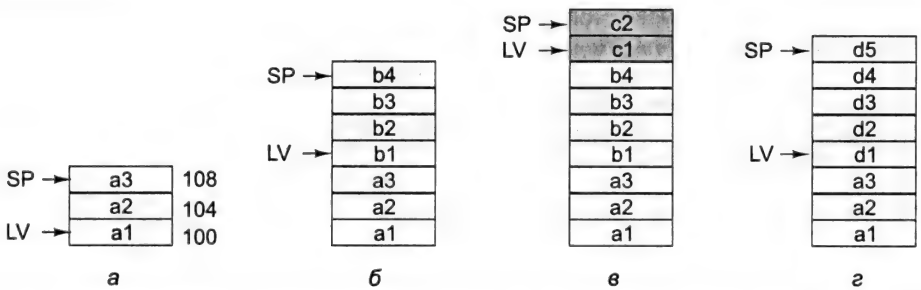


Рис. 4.7. Стек для хранения локальных переменных: во время процедуры *A* (*а*); после того как процедура *A* вызывает процедуру *B* (*б*); после того как процедура *B* вызывает процедуру *C* (*в*); после того как процедуры *C* и *B* прекращаются, а процедура *A* вызывает процедуру *D* (*г*)

А теперь давайте посмотрим, что происходит, если процедура *A* вызывает другую процедуру, *B*. Где должны храниться 4 локальные переменные процедуры *B* (*b1*, *b2*, *b3*, *b4*)? Ответ: в стеке, расположенном над стеком для процедуры *A*, как показано на рис. 4.7, *б*. Отметим, что после вызова процедуры регистр LV указывает уже на локальные переменные процедуры *B*. Обращаться к локальным переменным процедуры *B* можно по их сдвигу от LV. Если процедура *B* вызывает про-

цедуру *C*, то регистры LV и SP снова переопределяются и указывают на местонахождение локальных переменных процедуры *C*, как показано на рис. 4.7, *в*.

Когда процедура *C* завершается, *B* снова активизируется и стек возвращается в прежнее состояние (см. рис. 4.7, *б*), так что LV теперь указывает на локальные переменные процедуры *B*. Когда процедура *B* завершается, стек переходит в исходное состояние (см. рис. 4.7, *а*). При любых условиях LV указывает на базовый адрес стекового фрейма для текущей процедуры, а SP — на верхнее слово этого фрейма.

Предположим, что процедура *A* вызывает процедуру *D*, которая содержит 5 локальных переменных. Соответствующий стек показан на рис. 4.7, *г*. Локальные переменные процедуры *D* используют участок памяти процедуры *B* и часть стека процедуры *C*. В памяти с такой организацией размещаются только текущие процедуры. Когда процедура завершена, отведенный для нее участок памяти освобождается.

Но стек используется не только для хранения локальных переменных, а также и для хранения операндов во время вычисления арифметических выражений. Такой стек называется **стеком операндов**. Предположим, что перед вызовом процедуры *B* процедура *A* должна произвести следующее вычисление:

a1=a2+a3

Чтобы вычислить эту сумму, можно поместить *a2* в стек, как показано на рис. 4.8, *а*. Тогда значение регистра SP увеличится на число, равное количеству байтов в слове (скажем, на 4), и будет указывать на адрес первого операнда. Затем в стек помещается переменная *a3*, как показано на рис. 4.8, *б*. Отметим, что названия процедур и переменных выбираются пользователем, а названия регистров и кодов операций встроены. Названия процедур и переменных мы выделяем в тексте курсивом.

Теперь можно произвести вычисление, выполнив команду, которая выталкивает два слова из стека, складывает их и помещает результат обратно в стек, как показано на рис. 4.8, *в*. После этого верхнее слово можно вытолкнуть из стека и поместить его в локальную переменную *a1*, как показано на рис. 4.8, *г*.

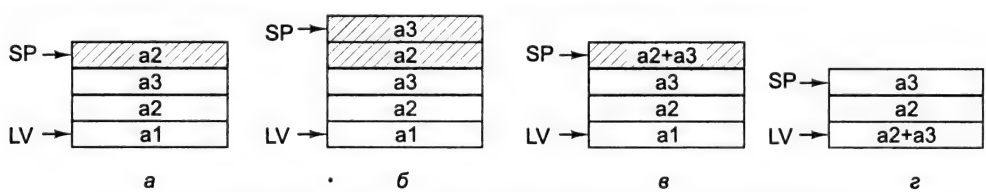


Рис. 4.8. Использование стека операндов для арифметических действий

Фреймы локальных переменных и стеки операндов могут смешиваться. Например, когда вызывается функция f при вычислении выражения $x^2+f(x)$, часть этого выражения (x^2) может находиться в стеке операндов. Результат вычисления функции остается в стеке над x^2, чтобы следующая команда сложила их.

Следует упомянуть, что все машины используют стек для хранения локальных переменных, но не все используют его для операндов. В большинстве машин нет стека операндов, но у JVM и IJVM он есть. Стековые операции мы рассмотрим подробно в главе 5.

Модель памяти IJVM

А теперь мы можем рассмотреть архитектуру IJVM. Она состоит из памяти, которую можно рассматривать либо как массив из 4 294 967 296 байтов (4 Гбайт), либо как массив из 1 073 741 824 слов, каждое из которых содержит 4 байта. В отличие от большинства архитектур команд, виртуальная машина Java не совершает обращений к памяти, видимых на уровне команд, но здесь существует несколько неявных адресов, которые составляют основу для указателя. Команды IJVM могут обращаться к памяти только через эти указатели. Определены следующие области памяти:

1. *Набор констант.* Эта область состоит из констант, цепочек и указателей на другие области памяти, на которые можно делать ссылку. Данная область загружается в тот момент, когда программа загружается из памяти, и после этого не меняется. Существует неявный регистр CPP (Constant Pool Pointer — указатель набора констант), который содержит адрес первого слова набора констант.
2. *Фрейм локальных переменных.* Эта область предназначена для хранения переменных во время выполнения процедуры. Она называется **фреймом локальных переменных**. В начале этого фрейма располагаются параметры (или аргументы) вызванной процедуры. Фрейм локальных переменных не включает в себя стек операндов. Он помещается отдельно. Исходя из соображений производительности, мы поместили стек операндов прямо над фреймом локальных переменных. Существует неявный регистр, который содержит адрес первой переменной фрейма. Мы назовем этот регистр LV (Local Variable — локальная переменная). Параметры вызванной процедуры хранятся в начале фрейма локальных переменных.
3. *Стек операндов.* Стек операндов не должен превышать определенный размер, который заранее вычисляется компилятором Java. Пространство стека операндов располагается прямо над фреймом локальных переменных, как показано на рис. 4.9. В данном случае стек операндов удобно считать частью фрейма локальных переменных. В любом случае существует виртуальный регистр, который содержит адрес верхнего слова стека. Отметим, что в отличие от регистров CPP и LV этот указатель меняется во время выполнения процедуры, поскольку операнды помещаются в стек и выталкиваются из него.
4. *Область процедур.* Наконец, существует область памяти, в которой содержится программа. Есть виртуальный регистр, содержащий адрес команды, которая будет вызвана следующей. Этот указатель называется счетчиком команд, или PC (Program Counter). В отличие от других участков памяти, область процедуры представляет собой массив байтов.

Следует сделать одно примечание по поводу указателей. Регистры CPP, LV и SP указывают на *слова*, а не на *байты*, и смещения происходят на определенное число слов. Например, LV, LV+1 и LV+2 указывают на первые три слова из фрейма локальных переменных, а LV, LV+4 и LV+8 — на слова, расположенные на расстоянии четырех слов (16 байтов) друг от друга.

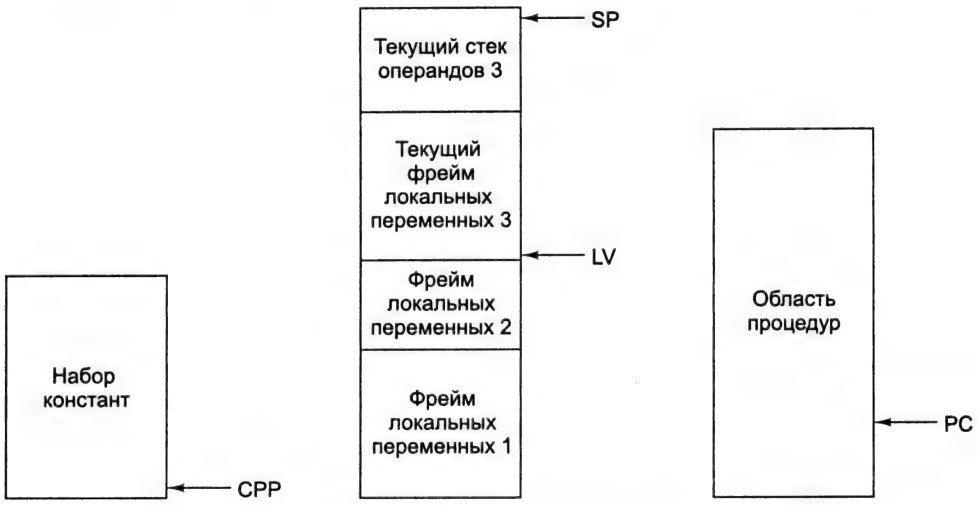


Рис. 4.9. Области памяти IJVM

Регистр PC, напротив, содержит адреса байтов, и изменение этого значения означает увеличение на определенное количество байтов, а не слов. Обращение к памяти регистром PC отличается от обращений других регистров, поэтому в машине Mic-1 и предусмотрен специальный порт памяти для PC. Запомните, что его размер составляет всего один байт. Если увеличить PC на единицу и начать процесс чтения, то это приведет к вызову следующего *байта*. Если увеличить SP на единицу и начать процесс чтения, то это приведет к вызову следующего *слова*.

Набор команд IJVM

Набор команд IJVM приведен в табл. 4.2. Каждая команда состоит из кода операции и иногда из операнда (например, смещения адреса или константы). В первом столбце приводится шестнадцатеричный код команды. Во втором столбце дается мнемоника языка ассемблера. В третьем столбце описывается предназначение команды.

Команды нужны для того, чтобы помещать слова из различных источников в стек. Источники — это набор констант (`LDC_W`), фрейм локальных переменных (`ILOAD`) и сама команда (`BIPUSH`). Переменную можно также вытолкнуть из стека и сохранить ее во фрейме локальных переменных (`ISTORE`). Над двумя верхними словами стека можно совершать две арифметические (`IADD` и `ISUB`) и две логические операции (`IAND` и `IOR`). При выполнении любой арифметической или логической операции два слова выталкиваются из стека, а результат помещается обратно в стек. Существует 4 команды перехода: одна для безусловного перехода (`GOTO`), а три другие для условных переходов (`IFEQ`, `IFLT` и `IF_ICMPEQ`). Все эти команды изменяют значение PC на размер их смещения, который следует за кодом операции в команде. Операнд смещения состоит из 16 битов. Он прибавляется к адресу кода операции. Существуют также команды для перестановки двух верхних слов стека (`SWAP`), дублирования верхнего слова (`DUP`) и удаления верхнего слова (`POP`).

Таблица 4.2. Набор команд IJVM. Размер операндов byte, const и varnum — 1 байт. Размер операндов disp, index и offset — 2 байта

Число	Мнемоника	Примечание
0x10	BIPUSH *byte*	Помещает байт в стек
0x59	DUP	Копирует верхнее слова стека и помещает его в стек
0xA7	GOTO *offset*	Безусловный переход
0x60	IADD	Выталкивает два слова из стека; помещает в стек их сумму
0x7E	IAND	Выталкивает два слова из стека; помещает в стек результат логического умножения (операция И)
0x99	IFEQ *offset*	Выталкивает слово из стека и совершает переход, если оно равно нулю
0x9B	IFLT *offset*	Выталкивает слово из стека и совершает переход, если оно меньше нуля
0x9F	IF_ICMPEQ *offset*	Выталкивает два слова из стека; совершает переход, если они равны
0x84	IINC *varnum const*	Прибавляет константу к локальной переменной
0x15	ILOAD *varnum*	Помещает локальную переменную в стек
0xB6	INVOKEVIRTUAL *disp*	Вызывает процедуру
0x80	IOR	Выталкивает два слова из стека; помещает в стек результат логического сложения (операция ИЛИ)
0xAC	IRETURN	Выдает результат выполнения процедуры (целое число)
0x36	ISTORE *varnum*	Выталкивает слово из стека и запоминает его во фрейме локальных переменных
0x64	ISUB	Выталкивает два слова из стека; помещает в стек их разность
0x13	LDC_W *index*	Берет константу из набора констант и помещает ее в стек
0x00	NOP	Не производит никаких действий
0x57	POP	Удаляет верхнее слово стека
0x5F	SWAP	Переставляет два верхних слова стека
0xC4	WIDE	Префиксная команда; следующая команда содержит 16-битный индекс

Некоторые команды имеют сложный формат, допускающий краткую форму для часто используемых версий. Из всех механизмов, которые JVM применяет для этого, в IJVM мы включили два. В одном случае мы пропустили краткую форму в пользу более традиционной. В другом случае мы показываем, как префиксная команда `WIDE` может использоваться для изменения следующей команды.

Наконец, существует команда для вызова другой процедуры (`INVOKEVIRTUAL`) и команда для выхода из текущей процедуры и возвращения к процедуре, из которой она была вызвана. Из-за сложности механизма мы немного упростили определение. Ограничение состоит в том, что, в отличие от языка Java, в нашем примере процедура может вызывать только такую процедуру, которая находится внутри нее. Это ограничение сильно искажает язык Java, но зато позволяет представить более простой механизм, избегая требования размещать процедуру динамически. (Если вы не знакомы с объектно-ориентированным программированием, вы можете пропустить это предложение. Мы просто превратили язык Java из объектно-

ориентированного в обычный, такой как C или Pascal.) На всех компьютерах, кроме JVM, адрес процедуры, которую нужно вызвать, непосредственно определяется командой CALL, поэтому наш подход скорее правило, чем исключение.

Механизм вызова процедуры состоит в следующем. Сначала вызывающая программа помещает в стек указатель на объект, который нужно вызвать. На рис. 4.10, *а* этот указатель обозначен буквами OBJREF. Затем вызывающая программа помещает в стек параметры процедуры (в данном примере *Параметр 1*, *Параметр 2* и *Параметр 3*). После этого выполняется команда INVOKEVIRTUAL.

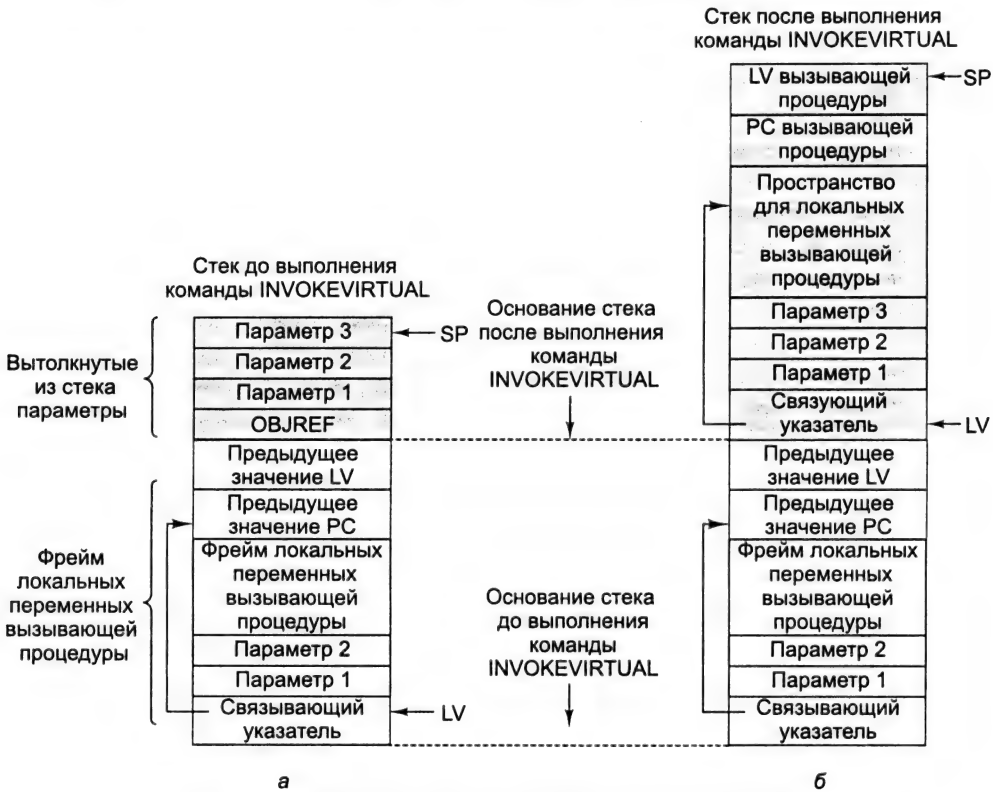


Рис. 4.10. Память до выполнения команды INVOKEVIRTUAL (*а*); память после выполнения этой команды (*б*)

Команда INVOKEVIRTUAL включает в себя относительный адрес (*disp*). Он указывает на позицию в наборе констант. В этой позиции содержится начальный адрес вызываемой процедуры, которая хранится в области процедур. Первые 4 байта в области процедур содержат специальные данные. Первые два байта представляют собой целое 16-битное число, указывающее на количество параметров данной процедуры (сами параметры были ранее помещены в стек). В данном случае OBJREF считается параметром: параметром 0. Это 16-битное целое число вместе со значением SP дает адрес OBJREF. Отметим, что регистр LV указывает на OBJREF, а не

на первый реальный параметр. Выбор, на что указывает LV, в какой-то степени произволен.

Следующие два байта в области процедур представляют еще одно 16-битное целое число, указывающее размер области локальных переменных для вызываемой процедуры. Дело в том, что для данной процедуры предоставляется новый стек, который размещается прямо над фреймом локальных переменных, для этого и нужно это число. Наконец, пятый байт в области процедур содержит код первой операции, которую нужно выполнить.

Ниже описывается, что происходит перед вызовом процедуры (см. также рис. 4.10). Два байта без знака, которые следуют за кодом операции, используются для индексирования таблицы констант (первый байт — это старший байт). Команда вычисляет базовый адрес нового фрейма локальных переменных. Для этого из указателя стека вычитается число параметров, а LV устанавливается на OBJREF. В OBJREF хранится адрес ячейки, в которой находится старое значение PC. Этот адрес вычисляется следующим образом. К размеру фрейма локальных переменных (параметры + локальные переменные) прибавляется адрес, содержащийся в регистре LV. Сразу над адресом, в котором должно быть сохранено старое значение PC, находится адрес, в котором должно быть сохранено старое значение LV. Над этим адресом начинается стек для новой вызванной процедуры. SP указывает на старое значение LV, адрес которого находится сразу под первой пустой ячейкой стека. Помните, что SP всегда указывает на верхнее слово в стеке. Если стек пуст, то SP указывает на адрес, который находится непосредственно под стеком, поскольку стек заполняется снизу вверх.

И наконец, для выполнения команды `INVOKEVIRTUAL` нужно сделать так, чтобы PC указывал на пятый байт в кодовом пространстве процедуры.

Команда `IRETURN` противоположна команде `INVOKEVIRTUAL` (рис. 4.11). Она освобождает пространство, используемое процедурой. Она также возвращает стек в предыдущее состояние, за исключением того, что: 1) OBJREF и все параметры удаляются из стека; 2) возвращенное значение помещается в стек, туда, где раньше находился OBJREF. Чтобы восстановить прежнее состояние, команда `IRETURN` должна вернуть прежние значения указателей PC и LV. Для этого она обращается к связующему указателю (это слово, определяемое текущим значением LV). В этом месте, где изначально находился параметр OBJREF, команда OBJREF сохранила адрес, содержащий старое значение PC. Это слово и слово над ним извлекаются, чтобы восстановить старые значения PC и LV соответственно. Возвращенное значение, которое хранится на самой вершине стека завершающейся процедуры, копируется туда, где изначально находился OBJREF, и теперь SP указывает на этот адрес. И тогда управление переходит к команде, которая следует сразу за `INVOKEVIRTUAL`.

До сих пор у нашей машины не было никаких команд ввода-вывода. Мы и не собираемся их вводить. В нашем примере, как и в виртуальной машине Java, они не нужны, и в описании JVM никогда не упоминаются процессы ввода-вывода. Считается, что машина без ввода-вывода более надежна. (Чтение и запись осуществляются в JVM путем вызова специальных процедур.)

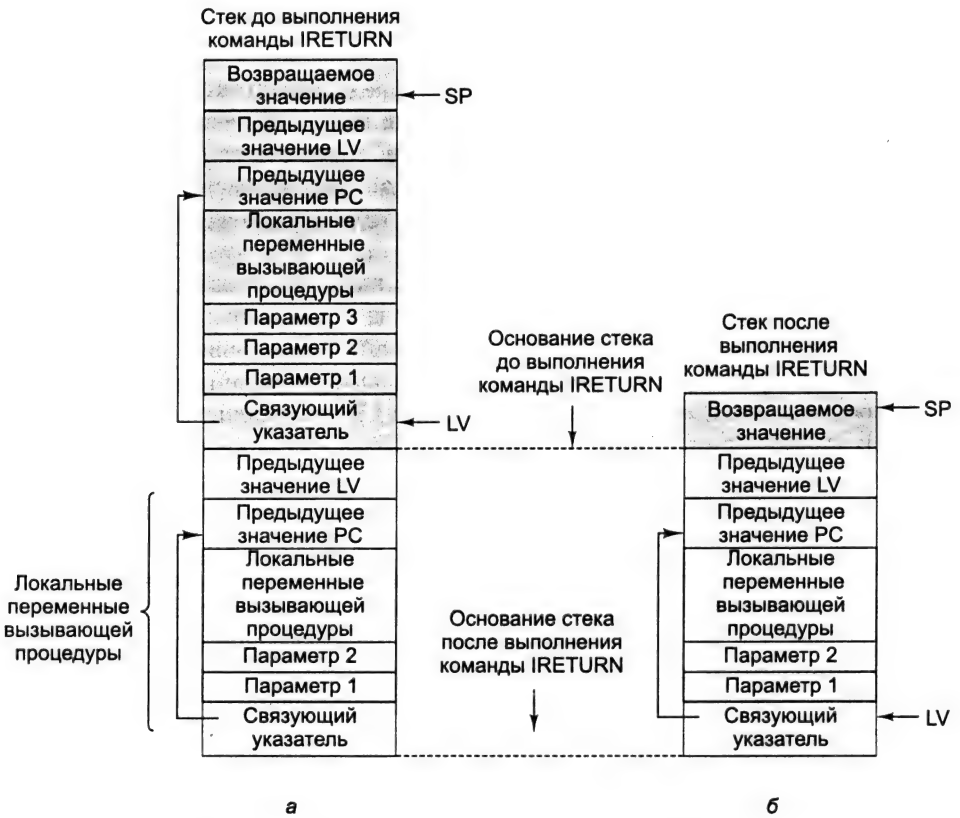


Рис. 4.11. Память до выполнения команды IRETURN (*а*); память после выполнения этой команды (*б*)

Компиляция Java для IJVM

А теперь рассмотрим, как язык Java связывается с машиной IJVM. В листинге 4.1 представлен небольшой фрагмент программы на языке Java. Компилятор Java должен был бы переделать эту программу в программу на языке ассемблер IJVM. Эта программа приведена в листинге 4.2. Цифры с 1 по 15 в левой части листинга, а также комментарии за значком «//» не являются частью самой программы. Они даны для наглядности и просто облегчают понимание. Затем ассемблер Java транслировал бы ее в программу в двоичном коде. Эта программа приведена в листинге 4.3. (В действительности компилятор Java сразу производит двоичную программу.) В данном примере i — локальная переменная 1, j — локальная переменная 2, а k — локальная переменная 3.

Листинг 4.1. Фрагмент программы на языке Java

```
i=j+k;
if(I==3)
   k=0;
else
   j=j-1;
```

Листинг 4.2. Программа на языке ассемблер Java

```
 1      ILOAD j         //i=j+k
 2      ILOAD k
 3      IADD
 4      ISTORE i
 5      ILOAD i         //if(i==3)
 6      BIPUSH 3
 7      IF_ICMPEQ L1
 8      ILOAD j         //j=j-1
 9      BIPUSH 1
10      ISUB
11      ISTORE j
12      GOTO L2
13  L1: BIPUSH 0       //k=0
14      ISTORE k
15  L2:
```

Листинг 4.3. Программа IJVM в шестнадцатеричном коде

```
0x15 0x02
0x15 0x03
0x60
0x36 0x01
0x15 0x01
0x10 0x03
0x9F 0x00 0x0D
0x15 0x02
0x10 0x01
0x64
0x36 0x02
0xA7 0x00 0x07
0x10 0x00
0x36 0x03
```

Скомпилированная программа проста. Сначала j и k помещаются в стек, складываются, а результат сохраняется в i. Затем i и константа 3 помещаются в стек и сравниваются. Если они равны, то совершается условный переход к *L1*, где k получает значение 0. Если они не равны, то выполняется часть программы после IF_ICMPEQ. После этого осуществляется переход к *L2*, где сливаются части `else` и `then`.

Стек операндов для программы IJVM, приведенной в листинге 4.2, изображен на рис. 4.12. До начала выполнения программы стек пуст, что показывает горизонтальная черта над цифрой 0. После выполнения первой команды `ILOAD` j помещается в стек. (См. на рисунке прямоугольник над цифрой 1.) Цифра 1 означает, что выполнена команда 1. После выполнения второй команды `ILOAD` в стеке оказываются уже два слова, как показано в прямоугольнике над цифрой 2. После выполнения команды `IADD` в стеке остается только одно слово, которое представляет собой сумму j+k. Когда верхнее слово выталкивается из стека и сохраняется в i, стек снова становится пустым.

Команда 5 (ILOAD) начинает оператор `if`. Эта команда помещает i в стек. Затем идет константа 3 (в команде 6). После сравнения стек снова становится пустым (7). Команда 8 является началом фрагмента `else`. Он продолжается вплоть до команды 12, когда совершается переход к метке *L2*.

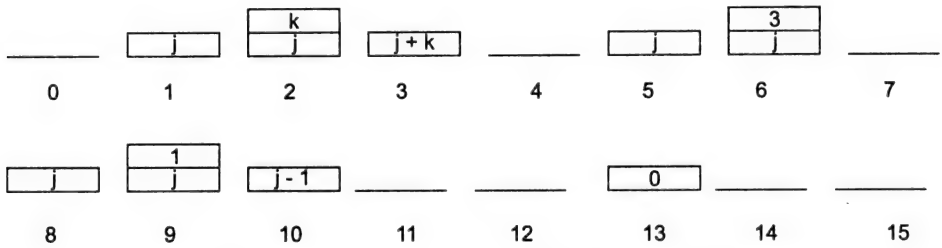


Рис. 4.12. Состояние стека после выполнения каждой команды в программе, приведенной в листинге 4.2

Пример реализации микроархитектуры

Мы подробно описали, что такое микроархитектура и макроархитектура. Осталось осуществить реализацию. Другими словами, нам предстоит узнать, что собой представляет и как работает программа микроархитектурного уровня, интерпретирующая команды макроархитектуры. Прежде чем ответить на эти вопросы, мы должны изложить систему обозначений, которую мы будем использовать для описания.

Микрокоманды и их запись

В принципе мы могли бы описать работу управляющей памяти с помощью двоичной системы счисления, по 36 битов в слове. Но гораздо удобнее ввести систему обозначений, с помощью которой можно передать суть рассматриваемых вопросов, и при этом не вдаваться в ненужные подробности. Важно понимать, что язык, который мы выбираем, предназначен для того, чтобы проиллюстрировать основные принципы работы программы, а не для того, чтобы использовать его в новых проектах. Если бы нашей целью было практическое применение языка, мы бы ввели совсем другую запись, чтобы максимально повысить гибкость программы. При этом была бы очень важна проблема выбора адресов, поскольку адреса в памяти не упорядочены. Насколько эффективным будет выбор адресов, зависит от способностей разработчика. Поэтому мы введем простой символический язык, который полностью описывает каждую операцию, но не объясняет полностью, как определяются все адреса.

Наша система обозначений показывает все действия, которые происходят на одной линии за один цикл. Теоретически для описания этих операций мы могли бы использовать язык высокого уровня. Однако контроль циклов очень важен, поскольку это дает возможность выполнять несколько операций одновременно. Кроме того, такой контроль необходим для того, чтобы можно было проанализировать каждый цикл, понять все операции и проверить их. Если целью разработки является повышение скорости и производительности, то имеет значение каждый цикл. При практической реализации в программу включается множество различных приемов для экономии циклов. В такой экономии есть большая выгода: четырехцикловая команда, которую можно сократить на два цикла, будет после этого выполняться в два раза быстрее. И такое повышение скорости достигается каждый раз, когда мы выполняем эту команду.

Один из возможных подходов — просто выдать список сигналов, которые должны активизироваться в каждом цикле. Предположим, что в одном цикле мы хотим

увеличить значение SP на единицу. Мы также хотим инициировать операцию чтения и хотим, чтобы следующая команда находилась в управляющей памяти в ячейке 122. Тогда мы могли бы написать:

```
ReadRegister=SP, ALU=INC, WSP, Read, NEXT_ADDRESS=122
```

Здесь WSP значит «записать регистр SP». Эта запись полная, но она сложна для понимания. Вместо этого мы соединим эти операции и передадим в записи результат действий:

```
SP=SP+1; rd
```

Назовем наш микроассемблер высокого уровня «MAL» (Micro Assembly Language — микроассемблер). По-французски «MAL» значит «болезнь» — это то, что с вами случится, если вы будете писать слишком большие программы на этом языке. Язык MAL разработан для того, чтобы продемонстрировать основные характеристики микроархитектуры. Во время каждого цикла могут записываться любые регистры, но обычно записывается только один. Значение только одного регистра может передаваться на шину B и в АЛУ. На шине A может быть +1, 0, –1 и регистр H. Следовательно, для обозначения определенной операции мы можем использовать простой оператор присваивания, как в языке Java. Например, чтобы копировать регистр SP в регистр MDR, мы можем написать:

```
MDR=SP
```

Чтобы показать, что мы используем какую-либо функцию АЛУ, мы можем написать, например:

```
MDR=H+SP
```

Эта строка означает, что значение регистра H складывается со значением регистра SP и результат записывается в регистр MDR. Операция сложения коммутативна (это значит, что порядок операндов не имеет значения), поэтому данное выше выражение можно записать в виде:

```
MDR=SP+H
```

и при этом породить ту же 36-битную микрокоманду, хотя, строго говоря, H является левым операндом АЛУ.

Мы должны использовать только допустимые операции. Самые важные операции приведены в табл. 4.3, где SOURCE — значение любого из регистров MDR, PC, MBR, MBRU, SP, LV, CPP, TOS и OPC (MBRU (MBR Unsigned) — это значение регистра MBR без знака). Все эти регистры могут выступать в качестве источников значений для АЛУ (они поступают в АЛУ через шину B). Сходным образом DEST может обозначать любой из следующих регистров: MAR, MDR, PC, SP, LV, CPP, TOS, OPC и H. Любой из этих регистров может быть пунктом назначения для выходного сигнала АЛУ, который передается к регистрам по шине C. Многие, казалось бы, разумные утверждения недопустимы. Например, выражение

```
MDR=SP+MDR
```

выглядит вполне корректно, но эту операцию нельзя выполнить в тракте данных, изображенном на рис. 4.5, за один цикл. Такое ограничение существует, поскольку для операции сложения (в отличие от увеличения или уменьшения на 1) один из операндов должен быть значением регистра H. Точно так же, выражение

```
H=H-MDR
```

могло бы пригодиться, но оно невозможно, поскольку единственным возможным источником вычитаемого является регистр H. Ассемблер должен отбрасывать выражения, которые кажутся пригодными, но в действительности недопустимы.

В нашей системе записи допускается использование нескольких операторов присваивания. Например, чтобы прибавить 1 к регистру SP и сохранить полученное значение в регистрах SP и MDR, нужно записать следующее:

```
SP=MDR=SP+1
```

Для обозначения процессов считывания из памяти и записи в память слов по 4 байта мы будем вставлять в микрокоманду слова `rd` и `wr`. Для вызова байта через 1-байтный порт используется команда `fetch`. Операции присваивания и операции взаимодействия с памятью могут происходить в одном и том же цикле. То, что происходит в одном цикле, записывается в одну строку.

Чтобы избежать путаницы, напомним еще раз, что Mic-1 может обращаться к памяти двумя способами. При чтении и записи 4-байтных слов данных используются регистры MAR/MDR. Эти процессы показываются в микрокомандах словами `rd` и `wr` соответственно. При чтении 1-байтных кодов операций из потока команд используются регистры PC/MBR. В микрокоманде это показывается словом `fetch`. Оба типа операций взаимодействия с памятью могут происходить одновременно.

Однако один и тот же регистр не может получать значение из памяти и тракта данных в одном и том же цикле. Рассмотрим кусок программы

```
MAR=SP; rd
MDR=H
```

В результате выполнения первой микрокоманды значение из памяти приписывается регистру MDR в конце второй микрокоманды. Однако вторая микрокоманда в то же самое время приписывает другое значение регистру MDR. Эти две операции присваивания конфликтуют, поскольку результаты не определены.

Таблица 4.3. Все допустимые операции. Любую из перечисленных операций можно расширить, добавив «<<8», что означает сдвиг результата влево на 1 байт. Например, часто используется операция H=MBR<<8

DEST=H
DEST=SOURCE
DEST=$\overline{H}$
DEST=$\overline{SOURCE}$
DEST=H+SOURCE
DEST=H+SOURCE+1
DEST=H+1
DEST=SOURCE+1
DEST=SOURCE–H
DEST=SOURCE–1
DEST= –H
DEST=H И SOURCE
DEST=H ИЛИ SOURCE
DEST=0
DEST=1
DEST= –1

Помните, что в каждой микрокоманде должен явно показываться адрес следующей микрокоманды. Однако часто бывает так, что микрокоманда вызывается только одной другой микрокомандой, а именно той микрокомандой, которая находится в строке над ней. Чтобы упростить работу программиста, микроассемблер обычно приписывает адрес каждой микрокоманде (порядок адресов может и не соответствовать последовательности микрокоманд в управляющей памяти) и заполняет поле NEXT_ADDRESS, так что последовательность выполнения микрокоманд соответствует последовательности строк микропрограммы.

Однако программисту иногда нужно совершить переход, условный или безусловный. Запись безусловных переходов проста:

```
goto label
```

Такая запись может включаться в любую микрокоманду. В ней явным образом указывается имя следующей микрокоманды. Например, очень часто последовательность микрокоманд заканчивается возвращением к первой команде основного цикла, поэтому последняя команда в каждой такой последовательности содержит запись

```
goto Main1
```

Отметим, что в тракте данных происходят обычные операции даже во время выполнения микрокоманд, которые содержат `goto`. В любой микрокоманде есть поле NEXT_ADDRESS. Команда `goto` сообщает ассемблеру, что в это поле вместо адреса микрокоманды, записанной в следующей строке, нужно поместить особое значение. В принципе каждая строка должна содержать запись `goto`, но если нужный адрес — это адрес микрокоманды, записанной в следующей строке, `goto` может опускаться для удобства.

Для условных переходов нам требуется другая запись. Помните, что JAMN и JAMZ используют биты N и Z соответственно. Например, иногда нужно проверить, не равно ли значение регистра 0. Для этого можно было бы пропустить это значение через АЛУ, сохранив его после этого в том же регистре. Тогда мы бы написали:

```
TOS=TOS
```

Запись выглядит забавно, но выполняет необходимые действия (устанавливает триггер Z и записывает значение в регистре TOS). В целях удобочитаемости микропрограммы мы расширили язык MAL, добавив два новых воображаемых регистра N и Z, которым можно присваивать значения. Например, строка

```
Z=TOS
```

пропускает значение регистра TOS через АЛУ, устанавливая триггер Z (и N), но при этом не сохраняет значение ни в одном из регистров. Использование регистра Z или N в качестве пункта назначения показывает микроассемблеру, что нужно установить все биты в поле C (см. рис. 4.4) на 0. Тракт данных проходит обычный цикл, выполняются все обычные допустимые операции, но ни один из регистров не записывается. Не важно, где находится пункт назначения: в регистре N или в регистре Z. Микрокоманды, которые при этом порождает микроассемблер, одинаковы. Программисты, выбравшие не тот регистр, в наказание будут неделю работать на первом компьютере IBM PC с частотой 4,77 МГц.

Чтобы микроассемблер установил бит JAMZ, нужно написать следующее:

```
if(Z) goto L1; else goto L2
```

Поскольку аппаратное обеспечение требует, чтобы 8 младших битов этих адресов совпадали, задача микроассемблера состоит в том, чтобы присвоить им такие адреса. С другой стороны, *L2* может находиться в любом из младших 256 слов управляющей памяти, поэтому микроассемблер без труда найдет подходящую пару.

Часто эти два утверждения сочетаются. Например,

```
Z=TOS; if(Z) goto L1; else goto L2
```

В результате такой записи MAL породит микрокоманду, в которой значение регистра TOS пропускается через АЛУ, но при этом нигде не сохраняется, так что это значение устанавливает бит Z. Сразу после загрузки из АЛУ бит Z соединяется со старшим битом регистра MPC через схему ИЛИ, вследствие чего адрес следующей микрокоманды будет вызван или из *L2*, или из *L1*. Значение регистра MPC стабилизируется, и он сможет использовать его для вызова следующей микрокоманды.

Наконец, нам нужна специальная запись, чтобы использовать бит JMPC:

```
goto (MBR OR value)
```

Эта запись сообщает микроассемблеру, что нужно использовать *value* (значение) для поля NEXT_ADDRESS и установить бит JMPC, так чтобы MBR соединялся через схему ИЛИ с регистром MPC вместе со значением NEXT_ADDRESS. Если *value* равно 0, достаточно написать:

```
goto (MBR)
```

Отметим, что только 8 младших битов регистра MBR соединяются с регистром MPC (см. рис. 4.5), поэтому вопрос о знаковом расширении тут не возникает. Также отметим, что используется то значение MBR, которое доступно в конце текущего цикла.

Реализация IJVM с использованием Mic-1

Сейчас мы уже дошли до того момента, когда можно соединить все части вместе. В табл. 4.4 приводится микропрограмма, которая работает на микроархитектуре Mic-1 и интерпретирует IJVM. Программа очень короткая — всего 112 микрокоманд. Таблица состоит из трех столбцов. В первом столбце записано символическое обозначение микрокоманды, во втором — сама микрокоманда, а в третьем — комментарий. Как мы уже говорили, последовательность микрокоманд не обязательно соответствует последовательности адресов в управляющей памяти.

Выбор названий большинства регистров, изображенных на рис. 4.1, должен стать очевидным. Регистры CPP (Constant Pool Pointer — указатель набора констант), LV (Local Variable pointer — указатель фрейма локальных переменных) и SP (Stack Pointer — указатель стека) содержат указатели адресов набора констант, фрейма локальных переменных и верхнего элемента в стеке соответственно, а регистр PC (Program Counter — счетчик команд) содержит адрес байта, который нужно вызвать следующим из потока команд. Регистр MBR (Memory Buffer Register — буферный регистр памяти) — это 1-байтовый регистр, который содержит байты потока команд, поступающих из памяти для интерпретации. TOS и OPC — дополнительные регистры. Они будут описаны ниже.

В определенные моменты в каждом из этих регистров обязательно находится определенное значение. Однако каждый из них также может использоваться в качестве временного регистра в случае необходимости. В начале и конце каждой команды регистр TOS (Top Of Stack register — регистр вершины стека) содержит значение адреса памяти, на который указывает SP. Это значение избыточно, поскольку его всегда можно считать из памяти, но если хранить это значение в регистре, то обращение к памяти не требуется. Для некоторых команд использование регистра TOS, напротив, влечет за собой *больше* обращений к памяти. Например, команда POP отбрасывает верхнее слово стека и, следовательно, должна вызвать новое значение вершины стека из памяти и записать его в регистр TOS.

OPC — временный регистр. У него нет определенного заданного назначения. В нем, например, может храниться адрес кода операции для команды перехода, пока значение PC увеличивается, чтобы получить доступ к параметрам. Он также используется в качестве временного регистра в командах условного перехода.

Как и все интерпретаторы, микропрограмма, приведенная в табл. 4.4, включает в себя основной цикл, который вызывает, декодирует и выполняет команды интерпретируемой программы (в данном случае команды IJVM). Основной цикл начинается со строки `Main1`, а именно с инварианта (утверждения), что в регистр PC уже загружен адрес ячейки памяти, в которой содержится код операции. Более того, этот код операции уже вызван из памяти в регистр MBR. Когда мы вернемся к этой ячейке, мы должны быть уверены, что значение PC уже обновлено и указывает на код следующей операции, а сам код операции уже вызван из памяти в MBR.

Такая последовательность действий имеет место в начале каждой команды, поэтому важно сделать ее как можно более короткой. Разрабатывая аппаратное и программное обеспечение микроархитектуры Mic-1, мы смогли сократить основной цикл до одной микрокоманды. Каждый раз, когда выполняется эта микрокоманда, код операции, которую нужно выполнить, уже находится в регистре MBR. Эта микрокоманда, во-первых, осуществляет переход к микрокоду, который выполняет данную операцию, а во-вторых, вызывает следующий после кода операции байт, который может быть либо операндом, либо кодом операции.

А теперь мы можем объяснить главную причину, почему в каждой микрокоманде явным образом указывается следующая микрокоманда и почему последовательность команд может и не соответствовать порядку их расположения в памяти. Все адреса управляющей памяти, соответствующие кодам операций, должны быть зарезервированы для первого слова интерпретатора соответствующей команды. Так, из табл. 4.2 мы видим, что программа, которая интерпретирует команду `POP`, начинается в ячейке 0x57, а программа, которая интерпретирует команду `DUP`, начинается в ячейке 0x59. (Как язык MAL узнает, что команду POP нужно поместить в ячейку 0x57, — одна из загадок Вселенной. Предположительно, где-то существует файл, который сообщает ему об этом.)

К сожалению, программа, интерпретирующая команду `POP`, включает в себя три микрокоманды, поэтому если их расположить в памяти последовательно, то эта программа смешается с началом команды `DUP`. Поскольку все адреса управляющей памяти, которые соответствуют кодам операций, зарезервированы, то все микрокоманды, идущие после первой микрокоманды в каждой последовательности, должны размещаться в промежутках между зарезервированными адресами. По этой

причине происходит очень много «скачков», и было бы нерационально каждый раз вставлять микрокоманду перехода, чтобы перепрыгнуть от одной последовательности адресов к другой.

Чтобы понять, как работает интерпретатор, предположим, что регистр MBR содержит значение 0x60, то есть код операции IADD (см. табл. 4.2). В основном цикле, который состоит из одной микрокоманды, выполняется следующее:

1. Значение регистра PC увеличивается, и теперь он содержит адрес первого байта после кода операции.
2. Начинается передача следующего байта в регистр MBR. Этот байт понадобится рано или поздно либо в качестве операнда текущей команды IJVM, либо в качестве кода следующей операции (как в случае с командой IADD, у которой нет операндов).
3. Совершается переход к адресу, который содержался в регистре MBR в начале цикла Main 1. Номер адреса равен значению кода операции, которая выполняется в данный момент. Этот адрес был помещен туда предыдущей микрокомандой. Отметим, что значение, которое вызывается из памяти во время этой микрокоманды, не играет никакой роли в межуровневом переходе.

Здесь начинается вызов следующего байта, поэтому он будет доступен уже к концу третьей микрокоманды. В этот момент он, может быть, и не нужен, но он в любом случае когда-нибудь понадобится, поэтому не будет никакого вреда в том, что вызов начнется именно здесь.

Таблица 4.4. Микропрограмма для Mic-1

Микрокоманда	Операции	Комментарий
Main1	PC=PC+1; fetch; goto(MBR)	MBR содержит код операции; получение следующего байта; отсылка
nop1	goto Main1	Ничего не происходит
iadd1	MAR=SP=SP–1; rd	Чтение слова, идущего после верхнего слова стека
iadd2	H=TOS	H = вершина стека
iadd3	MDR=TOS=MDR+H;wr; goto Main1	Суммирование двух верхних слов; запись суммы в верхнюю позицию стека
isub1	MAR=SP=SP–1; rd	Чтение слова, идущего после верхнего слова стека
isub2	H=TOS	H = вершина стека
isub3	MDR=TOS=MDR–H;wr; goto Main1	Вычитание; запись результата в вершину стека
iand1	MAR=SP=SP–1; rd	Чтение слова, идущего после верхнего слова стека
iand2	H=TOS	H = вершина стека
iand3	MDR=TOS=MDR И H;wr; goto Main1	Операция И; запись результата в вершину стека
ior1	MAR=SP=SP–1; rd	Чтение слова, идущего после верхнего слова стека
ior2	H=TOS	H = вершина стека
ior3	MDR=TOS=MDR ИЛИ H; wr;goto Main1	Операция ИЛИ; запись результата в вершину стека
dup1	MAR=SP=SP+1	Увеличение SP на 1 и копирование результата в регистр MAR

Микрокоманда	Операции	Комментарий
dup2	MDR=TOS; wr; goto Main1	Запись нового слова в стек
pop1	MAR=SP=SP–1; rd	Чтение слова, идущего после верхнего слова стека
pop2		Программа ждет, пока считается из памяти новое значение регистра TOS
pop3	TOS=MDR; goto Main1	Копирование нового слова в регистр TOS
swap1	MAR=SP=SP–1; rd	Установка регистра MAR на значение SP–1; чтение второго слова из стека
swap2	MAR=SP	Установка регистра MAR на верхнее слово стека
swap3	H=MDR; wr	Сохранение значения TOS в регистре H; запись второго слова в вершину стека
swap4	MDR=TOS	Копирование прежнего значения TOS в регистр MDR
swap5	MAR=SP–1; wr	Установка регистра MAR на значение SP–1; запись второго слова в стек
swap6	TOS=H; goto Main1	Обновление TOS
bipush1	SP=MAR=SP+1	MBR = байт, который нужно поместить в стек
bipush2	PC=PC+1; fetch	Увеличение PC на 1; вызов кода следующей операции
bipush3	MDR=TOS=MBR; wr; goto Main1	Добавление к байту дополнительного знакового разряда и запись значения в стек
iload1	H=LV	MBR содержит индекс; копирование значения LV в H
iload2	MAR=MBRU+H; rd	MAR = адрес локальной переменной, которую нужно поместить в стек
iload3	MAR=SP=SP+1	Регистр SP указывает на новую вершину стека; подготовка к записи
iload4	PC=PC+1; fetch; wr	Увеличение значения PC на 1; вызов кода следующей операции; запись вершины стека
iload5	TOS=MDR; goto Main1	Обновление TOS
istore1	H=LV	MBR содержит индекс; копирование значения LV в H
istore2	MAR=MBRU+H	MAR = адрес локальной переменной, в которую нужно сохранить слово из стека
istore3	MDR=TOS; wr	Копирование значения TOS в регистр MDR; запись слова
istore4	SP=MAR=SP–1; rd	Чтение из стека второго слова сверху
istore5	PC=PC+1; fetch	Увеличение PC на 1; вызов следующего кода операции
istore6	TOS=MDR; goto Main1	Обновление TOS
wide1	PC=PC+1;fetch; goto(MBR ИЛИ 0x100)	Межуровневый переход к старшим адресам
wide_iload1	PC=PC+1;fetch	MBR содержит первый байт индекса; вызов второго байта
wide_iload2	H=MBRU<<8	H = первый байт индекса, сдвинутый влево на 8 битов
wide_iload3	H=MBRU ИЛИ H	H = 16-битный индекс локальной переменной
wide_iload4	MAR=LV+H; rd; goto iload3	MAR = адрес локальной переменной, которую нужно записать в стек

продолжение

Таблица 4.4 (*продолжение*)

Микрокоманда	Операции	Комментарий
wide_istore1	PC=PC+1;fetch	MBR содержит первый байт индекса; вызов второго байта
wide_istore2	H=MBRU<<8	H = первый байт индекса, сдвинутый влево на 8 битов
wide_istore3	H=MBRU ИЛИ H	H = 16-битный индекс локальной переменной
wide_istore4	MAR=LV+H; rd; goto istore3	MAR = адрес локальной переменной, в которую нужно записать слово из стека
ldc_w1	PC=PC+1;fetch	MBR содержит первый байт индекса; вызов второго байта
ldc_w2	H=MBRU<<8	H = первый байт индекса, сдвинутый влево на 8 битов
ldc_w3	H=MBRU ИЛИ H	H = 16-битный индекс константы в наборе констант
ldc_w4	MAR=H+CPP; rd; goto iload3	MAR = адрес константы в наборе констант
iinc1	H=LV	MBR содержит индекс; копирование значения LV в H
iinc2	MAR=MBRU+H; rd	Копирование суммы значения LV и индекса в регистр MAR; чтение переменной
iinc3	PC=PC+1; fetch	Вызов константы
iinc4	H=MDR	Копирование переменной в регистр H
iinc5	PC=PC+1; fetch	Вызов следующего кода операции
iinc6	MDR=MBR+H; wr; goto Main1	Запись суммы в регистр MDR; обновление переменной
goto1	OPC=PC–1	Сохранение адреса кода операции
goto2	PC=PC+1; fetch	MBR = первый байт смещения; вызов второго байта
goto3	H=MBR<<8	Сдвиг первого байта влево на 8 битов и сохранение его в регистре H
goto4	H=MBRU ИЛИ H	H =16-битное смещение перехода
goto5	PC=OPC+H; fetch	Суммирование смещения и OPC
goto6	goto Main1	Ожидание вызова следующего кода операции
iflt1	MAR=SP=SP–1; rd	Чтение второго сверху слова в стеке
iflt2	OPC=TOS	Временное сохранение TOS в OPC
iflt3	TOS=MDR	Запись новой вершины стека в TOS
iflt4	N=OPC; if(N) goto T; else goto F	Переход в бит N
ifeq1	MAR=SP=SP–1; rd	Чтение второго сверху слова в стеке
ifeq2	OPC=TOS	Временное сохранение TOS в OPC
ifeq3	TOS=MDR	Запись новой вершины стека в TOS
ifeq4	Z=OPC; if(Z) goto T; else goto F	Переход в бит Z
if_icmpeq1	MAR=SP=SP–1; rd	Чтение второго сверху слова в стеке
if_icmpeq2	MAR=SP=SP–1	Установка регистра MAR на чтение новой вершины стека
if_icmpeq3	H=MDR; rd	Копирование второго слова из стека в регистр H

Микрокоманда	Операции	Комментарий
if_icmpeq4	OPC=TOS	Временное сохранение TOS в OPC
if_icmpeq5	TOS=MDR	Помещение новой вершины стека в TOS
if_icmpeq6	Z=OPC–H; if(Z) goto T; else goto F	Если два верхних слова равны, осуществляется переход к T; если они не равны, осуществляется переход к F
T	OPC=PC–1; fetch; goto goto2	То же, что goto1; нужно для адреса целевого объекта
F	PC=PC+1	Пропуск первого байта смещения
F2	PC=PC+1; fetch	PC указывает на следующий код операции
F3	goto Main1	Ожидание вызова кода операции
invoke_virtual1	PC=PC+1; fetch	MBR = первый байт индекса; увеличение PC на 1; вызов второго байта
invoke_virtual2	H=MBRU<<8	Сдвиг первого байта на 8 битов и сохранение значения в регистре H
invoke_virtual3	H=MBRU ИЛИ H	H = смещение указателя процедуры от регистра CPP
invoke_virtual4	MAR=CPP+H; rd	Вызов указателя процедуры из набора констант
invoke_virtual5	OPC=PC+1	Временное сохранение значения PC в регистре OPC
invoke_virtual6	PC=MDR; fetch	Регистр PC указывает на новую процедуру; вызов числа параметров
invoke_virtual7	PC=PC+1; fetch	Вызов второго байта числа параметров
invoke_virtual8	H=MBRU<<8	Сдвиг первого байта на 8 битов и сохранение значения в регистре H
invoke_virtual9	H=MBRU ИЛИ H	H = число параметров
invoke_virtual10	PC=PC+1; fetch	Вызов первого байта размера области локальных переменных
invoke_virtual11	TOS=SP–H	TOS = адрес OBJREF–1
invoke_virtual12	TOS=MAR=TOS+1	TOS = адрес OBJREF (новое значение LV)
invoke_virtual13	PC=PC+1; fetch	Вызов второго байта размера области локальных переменных
invoke_virtual14	H=MBRU<<8	Сдвиг первого байта на 8 битов и сохранение значения в регистре H
invoke_virtual15	H=MBRU ИЛИ H	H = размер области локальных переменных
invoke_virtual16	MDR=SP+H+1; wr	Перезапись OBJREF со связующим указателем
invoke_virtual17	MAR=SP=MDR	Установка регистров SP и MAR на адрес ячейки, в которой содержится старое значение PC
invoke_virtual18	MDR=OPC; wr	Сохранение старого значения PC над локальными переменными
invoke_virtual19	MAR=SP=SP+1	SP указывает на ячейку, в которой хранится старое значение LV
invoke_virtual20	MDR=LV; wr	Сохранение старого значения LV над сохраненным значением PC
invoke_virtual21	PC=PC+1; fetch	Вызов первого кода операции новой процедуры
invoke_virtual22	LV=TOS; goto Main1	Установка значения LV на первый адрес фрейма локальных переменных
ireturn1	MAR=SP=LV; rd	Переустановка регистров SP и MAR для вызова связующего указателя

продолжение

Таблица 4.4 (*продолжение*)

Микрокоманда	Операции	Комментарий
ireturn2		Процесс считывания
ireturn3	LV=MAR=MDR; rd	Установка регистра LV на связующий указатель; вызов старого значения PC
ireturn4	MAR=LV+1	Установка регистра MAR на чтение старого значения LV
ireturn5	PC=MDR; rd; fetch	Восстановление PC; вызов следующего кода операции
ireturn6	MAR=SP	Установка MAR на запись TOS
ireturn7	LV=MDR	Восстановление LV
ireturn8	MDR=TOS; wr; goto Main1	Сохранение результата в изначальной вершине стека

Если все разряды байта в регистре MBR равны 0 (это код операции для команды `NOP`), то следующей будет микрокоманда `nop1`, которая вызывается из ячейки 0. Поскольку эта команда не производит никаких операций, она просто совершает переход к началу основного цикла, где повторяется та же последовательность действий, но уже с новым кодом операции в MBR.

Еще раз подчеркнем, что микрокоманды, приведенные в табл. 4.4, не расположены в памяти последовательно и что микрокоманда `Main1` находится вовсе не в ячейке с адресом 0 (поскольку в этой ячейке должна находиться микрокоманда nop1). Задача микроассемблера — поместить каждую команду в подходящую ячейку и связать их в короткие последовательности, используя поле NEXT_ADDRESS. Каждая последовательность начинается с адреса, который соответствует номерному значению кода операции (например, команда `POP` начинается с адреса 0x57), но остальные части последовательности могут находиться в любых ячейках управляющей памяти, и эти ячейки не обязательно идут подряд.

А теперь рассмотрим команду `IADD`. Она начинается с микрокоманды `iadd1`. Требуется выполнить следующие действия:

1. Значение регистра TOS уже есть, но из памяти нужно вызвать второе слово стека.
2. Значение регистра TOS нужно прибавить ко второму слову стека, вызванному из памяти.
3. Результат, который помещается в стек, должен быть сохранен в памяти и в регистре TOS.

Для того чтобы вызвать операнд из памяти, необходимо уменьшить значение указателя стека и записать его в регистр MAR. Отметим, что этот адрес будет использоваться для последующей записи. Более того, поскольку эта ячейка памяти будет новой вершиной стека, данное значение должно быть присвоено регистру SP. Следовательно, определить новое значение SP и MAR, уменьшить значение SP на 1 и записать его в оба регистра можно за одну операцию.

Все эти действия выполняются в первом цикле (`iadd1`). Здесь же инициируется операция чтения. Кроме того, регистр MPC получает значение из поля NEXT ADDRESS микрокоманды `iadd1`. Это адрес микрокоманды `iadd2`. Затем `iadd2` счи-

тывается из управляющей памяти. Во втором цикле, пока происходит считывание операнда из памяти, мы копируем верхнее слово стека из TOS в H, где оно будет доступно для сложения, когда процесс считывания завершится.

В начале третьего цикла (`iadd3`) MDR содержит второе слагаемое, вызванное из памяти. В этом цикле оно прибавляется к значению регистра H, а результат сохраняется обратно в регистры MDR и TOS. Кроме того, начинается операция записи, в процессе которой новое верхнее слово стека сохраняется в памяти. В этом цикле команда `goto` приписывает адрес `Main1` регистру MPC, таким образом, мы возвращаемся к исходному пункту и можем начать выполнение следующей операции.

Если следующий код операции, который содержится в данный момент в регистре MBR, равен 0x64 (`ISUB`), то повторяется практически та же последовательность действий. После выполнения `Main1` управление передается микрокоманде с адресом 0x64 (`isub1`). За этой микрокомандой следуют `isub2`, `isub3`, а затем снова `Main1`. Единственное различие между этой и предыдущей последовательностью состоит в том, что в цикле `isub3` содержание регистра H не прибавляется к значению MDR, а вычитается из него.

Команда `IAND` идентична командам `IADD` и `ISUB`, только в данном случае два верхних слова стека подвергаются логическому умножению (операция И), а не складываются и не вычитаются. Нечто подобное происходит и во время выполнения команды `IOR`.

Если код операции соответствует командам `DUP`, `POP` или `SWAP`, то нужно использовать стек. Команда `DUP` дублирует верхнее слово стека. Поскольку значение этого слова уже находится в регистре TOS, нужно просто увеличить SP на 1. Теперь регистр SP указывает на новый адрес. В эту новую ячейку и записывается значение регистра TOS. Команда `POP` тоже достаточно проста: нужно только уменьшить значение SP на 1, чтобы отбросить верхнее слово стека. Однако теперь необходимо считать новое верхнее слово стека из памяти и записать его в регистр TOS. Наконец, команда `SWAP` меняет местами значения двух ячеек памяти, а именно два верхних слова стека. Регистр TOS уже содержит одно из этих значений, поэтому считывать его (значение) из памяти не нужно. Подробнее мы обсудим эту команду немного позже.

Команда `BIPUSH` сложнее предыдущих, поскольку за кодом операции следует байт, как показано на рис. 4.13. Этот байт представляет собой целое число со знаком. Этот байт, который уже был передан в регистр MBR во время микрокоманды `Main1`, нужно расширить до 32 битов (знаковое расширение) и скопировать его в регистр MDR. Наконец, значение SP увеличивается на 1 и копируется в MAR, что позволяет записать операнд на вершину стека. Этот операнд также должен копироваться в регистр TOS. Отметим, что значение регистра PC должно увеличиваться на 1, чтобы в микрокоманде `Main1` следующий код операции уже имелся в наличии.

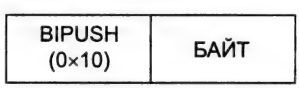


Рис. 4.13. Формат команды `BIPUSH`

Теперь рассмотрим команду `ILOAD`. В этой команде за кодом операции также следует байт (рис. 4.14, *а*), но этот байт представляет собой индекс (без знака),

используемый для того, чтобы найти в пространстве локальных переменных слово, которое нужно поместить в стек. Поскольку здесь имеется всего 1 байт, можно различать только 2^8=256 слов, а именно первые 256 слов пространства локальных переменных. Для выполнения команды ILOAD требуется и процесс чтения (чтобы вызвать слово), и процесс записи (чтобы поместить его в стек). Чтобы определить адрес для считывания, нужно прибавить смещение, которое хранится в регистре MBR (буферном регистре памяти), к содержимому регистра LV. Доступ к регистрам MBR и LV можно получить только через шину B, поэтому сначала значение LV копируется в регистр H (в цикле iload1), а затем прибавляется значение MBR. Результат суммирования копируется в регистр MAR, и начинается процесс чтения (в цикле iload2).

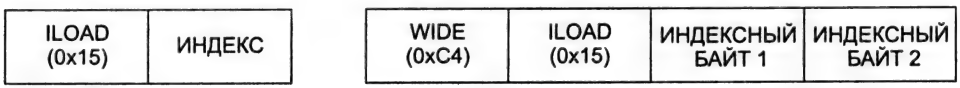


Рис. 4.14. Команда ILOAD с однобайтным индексом (*а*); команда WIDE ILOAD с двубайтным индексом (*б*)

Однако здесь регистр MBR используется не совсем так, как в команде BIPUSH, где байт расширен по знаку. В случае с индексом смещение всегда положительно, поэтому байт смещения должен быть целым числом без знака (в отличие от BIPUSH, где байт представляет собой 8-битное целое число со знаком). Интерфейс между регистром MBR и шиной B разработан таким образом, чтобы обе операции были возможны. В случае с BIPUSH (где байт — 8-битное целое число со знаком) самый левый бит значения MBR копируется в 24 старших бита шины B. В случае с ILOAD (где байт — 8-битное целое число без знака) 24 старших бита шины B заполняются нулями. Два специальных сигнала помогают определить, какую из этих двух операций нужно выполнить (см. рис. 4.5). В микропрограмме слово MBR указывает на байт со знаком (как в команде bipush3), а MBRU — на байт без знака (как в команде iload2).

Пока ожидается поступление операнда из памяти (во время iload3), значение регистра SP увеличивается на 1 для записи новой вершины стека. Это значение также копируется в регистр MAR (это требуется для записи операнда в стек). Затем значение PC снова увеличивается на 1, чтобы вызвать следующий код операции (микрокоманда iload4). Наконец, значение MDR копируется в регистр TOS, чтобы показать новое верхнее слово стека (микрокоманда iload5).

Команда ISTORE противоположна команде ILOAD (из стека выталкивается верхнее слово и сохраняется в ячейке памяти, адрес которой равен сумме значения регистра LV и индекса данной команды). В данном случае используется такой же формат, как и в команде ILOAD (рис. 4.14, *а*), только здесь код операции не 0x15, а 0x36. Поскольку верхнее слово стека уже известно (оно находится в регистре TOS), его можно сразу сохранить в памяти. Однако новое верхнее слово стека все же необходимо вызвать из памяти, поэтому требуется и операция чтения, и операция записи, хотя их можно выполнять в любом порядке (или даже одновременно, если бы это было возможно).

Команды ILOAD и ISTORE имеют доступ только к первым 256 локальным переменным. Хотя для большинства программ этого пространства будет достаточно, все же нужно иметь возможность обращаться к любой локальной переменной, в какой бы части фрейма она не находилась. Чтобы обеспечить такую возможность, машина IJVM использует то же средство, что и JVM: специальный код операции WIDE (так называемый **префиксный байт**), за которым следует код операции ILOAD или ISTORE. Когда встречается такая последовательность, формат команды ILOAD или ISTORE меняется, и за кодом операции идет не 8-битный, а 16-битный индекс, как показано на рис. 4.14, *б*.

Команда WIDE декодируется обычным способом. Сначала происходит переход к микрокоманде wide1, которая обрабатывает код операции WIDE. Хотя код операции, который нужно расширить, уже присутствует в регистре MBR, микрокоманда wide1 вызывает первый байт после кода операции, поскольку этого требует логика микропрограммы. Затем совершается еще один межуровневый переход, но на этот раз для перехода используется байт, который следует за WIDE. Но поскольку команда WIDE ILOAD требует набора микрокоманд, отличного от ILOAD, а команда WIDE ISTORE требует набора микрокоманд, отличного от ISTORE, и т. д., при осуществлении межуровневого перехода нельзя использовать в качестве целевого адреса код операции.

Вместо этого микрокоманда wide1 подвергает логическому сложению адрес 0x100 и код операции, поместив его в регистр MPC. В результате интерпретация WIDE LOAD начинается с адреса 0x115 (а не 0x15), интерпретация WIDE ISTORE — с адреса 0x136 (а не 0x36) и т. д. Таким образом, каждый код операции WIDE начинается с адреса, который в управляющей памяти на 256 (то есть 0x100) слов выше, чем соответствующий обычный код операции. Начальная последовательность микрокоманд для ILOAD и WIDE ILOAD показана на рис. 4.15.

Команда WIDE ILOAD отличается от обычной команды ILOAD только тем, что индекс в ней состоит из двух индексных байтов. Слияние и последующее суммирование этих байтов должно происходить по стадиям, при этом сначала первый индексный байт сдвигается влево на 8 битов и копируется в H. Поскольку индекс — целое число без знака, то здесь используется регистр MBRU (24 старших бита заполняются нулями). Затем прибавляется второй байт индекса (операция сложения идентична слиянию, поскольку младший байт регистра H в данный момент равен 0), при этом гарантируется, что между байтами не будет переноса. Результат снова сохраняется в регистре H. С этого момента происходят те же действия, что и в стандартной команде ILOAD. Вместо того чтобы дублировать последние команды ILOAD (от iload3 до iload5), мы просто совершили переход от wide_iload4 к iload3. Отметим, что во время выполнения этой команды значение PC должно увеличиваться на 1 дважды, чтобы в конце этот регистр указывал на следующий код операции. Команда ILOAD увеличивает значение один раз; последовательность команд WIDE_ILOAD также увеличивает это значение один раз.

Такая же ситуация имеет место при выполнении WIDE_ISTORE. После выполнения первых четырех микрокоманд (от wide_istore1 до wide_istore4) последовательность действий та же, что и в команде ISTORE после первых двух микрокоманд, поэтому мы совершаем переход от wide_istore4 к istore3.

Далее мы рассмотрим команду LDC_W. Существует два отличия этой команды от ILOAD. Во-первых, она содержит 16-битное смещение без знака (как и расширенная версия ILOAD), а во-вторых, эта команда индексируется из регистра CPP, а не из

LV, поскольку она считывает значение из набора констант, а не из фрейма локальных переменных. (Существует еще и краткая форма этой команды — LDC, но мы не стали включать ее в машину IJVM, поскольку полная форма содержит в себе все варианты краткой формы, хотя при этом занимает 3 байта вместо 2.)

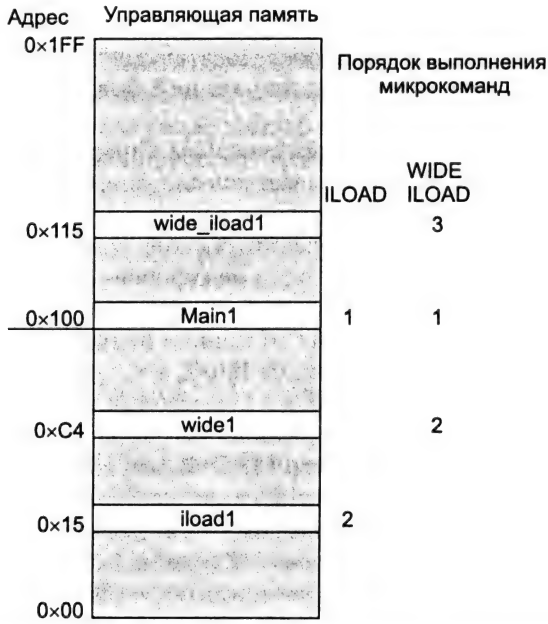


Рис. 4.15. Начало последовательности микрокоманд для ILOAD и WIDE ILOAD. Адреса приводятся в качестве примера

Команда IINC — единственная команда кроме ISTORE, которая может изменять локальную переменную. Она включает в себя два операнда по одному байту, как показано на рис. 4.16.



Рис. 4.16. Команда IINC содержит два поля операндов

Поле индекса нужно для того, чтобы определить смещение от начала фрейма локальных переменных. Команда считывает эту переменную, увеличивает ее на константу (константа содержится во втором поле) и помещает результат обратно в ту же ячейку памяти. Отметим, что константа является 8-битным числом со знаком в промежутке от −128 до +127. В машине JVM есть расширенная версия этой команды, в которой длина каждого операнда составляет два байта.

Рассмотрим первую команду перехода: GOTO. Эта команда изменяет значение регистра PC таким образом, чтобы следующая команда IJVM находилась в ячейке памяти с адресом, который вычисляется путем прибавления 16-битного смещения (со знаком) к адресу кода операции GOTO. Сложность здесь в том, что смещение связано с тем значением, которое содержится в регистре PC в начале декодирова-

ния команды, а не тем, которое содержится в том же регистре после вызова двух байтов смещения.

Чтобы лучше это понять, посмотрите на рис. 4.17, *а*. Здесь изображена ситуация, которая имеет место в начале цикла `Main1`. Код операции уже находится в регистре MBR, но значение PC еще не увеличилось. На рис. 4.17, *б* мы видим ситуацию в начале цикла `goto1`. В данном случае значение PC уже увеличено на 1 и первый байт смещения уже передан в MBR. В следующей микрокоманде (рис. 4.17, *в*) старое значение PC, которое указывает на код операции, сохраняется в регистре OPC. Это значение нам нужно, поскольку именно от него, а не от текущего значения PC, зависит смещение команды `GOTO`. И именно для этого предназначен регистр OPC.

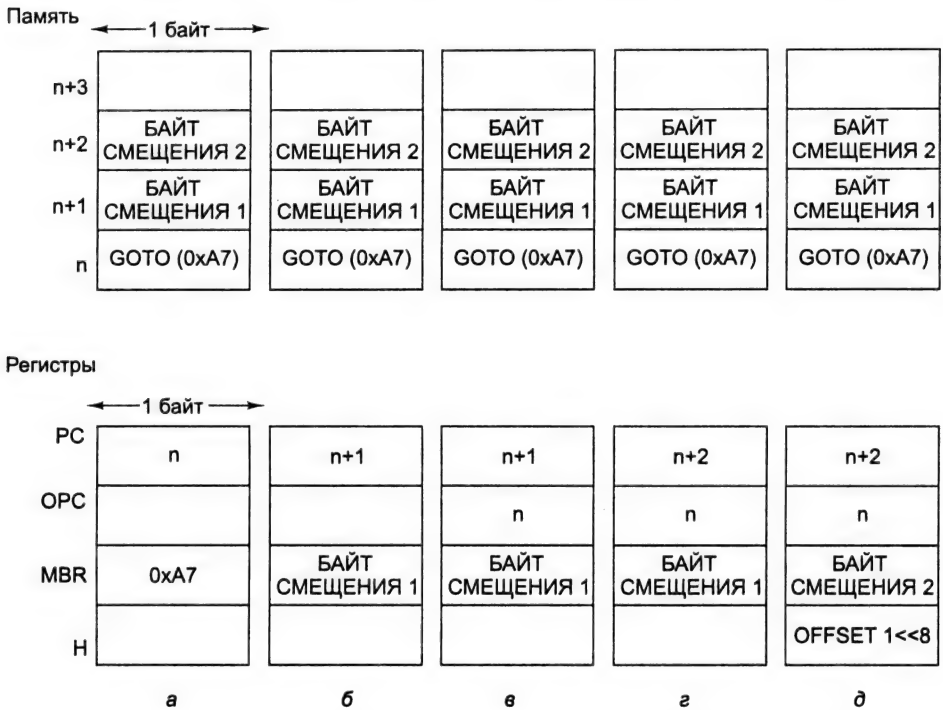


Рис. 4.17. Ситуация в начале выполнения различных микрокоманд: Main1 (*а*); goto1 (*б*); goto2 (*в*); goto3 (*г*); goto4 (*д*)

Микрокоманда `goto2` начинает вызов второго байта смещения, что приводит к ситуации, показанной на рис. 4.17, *г* (микрокоманда `goto3`). После того как первый байт смещения сдвигается влево на 8 битов и копируется в регистр H, мы переходим к микрокоманде `goto4` (см. рис. 4.17, *д*). Теперь у нас первый байт смещения, сдвинутый влево, находится в регистре H, второй байт смещения — в регистре MBR, а основа смещения — в регистре OPC. В микрокоманде `goto5` путем прибавления полного 16-битного смещения к основе смещения мы получаем новый адрес, который помещается в регистр PC. Отметим, что в `goto4` вместо MBR мы используем MBRU, поскольку нам не нужно знаковое расширение второго байта. 16-битное смещение строится путем логического сложения (операция ИЛИ) двух половинок. Нако-

нец, поскольку программа перед переходом к Main1 требует, чтобы в MBR был помещен следующий код операции, мы должны вызвать этот код. Последний цикл, goto6, нужен для того, чтобы вовремя помесить данные из памяти в регистр MBR.

Смещения, которые используются в команде goto, представляют собой 16-битные значения со знаком в промежутке от –32768 до +32767. Это значит, что переходы на более дальние расстояния невозможны. Это свойство можно рассматривать либо как дефект, либо как особенность машины IJVM (а также JVM). Те, кто считает это дефектом, скажут, что машина JVM не должна ограничивать возможности программирования. Те, кто считает это особенностью, скажут, что работа многих программистов продвинулась бы кардинальным образом, если бы им в ночных кошмарах снилось следующее сообщение компилятора:

```
Program is too big and hairy. You must rewrite it. Compilation aborted. (Программа
слишком длинная и сложная. Вы должны переписать ее. Компиляция прекращена.)
```

К сожалению (это наша точка зрения), это сообщение появляется только в том случае, если выражение else или then превышает 32 Кбайт, что составляет, по крайней мере, 50 страниц текста на языке Java.

А теперь рассмотрим три команды условного перехода: IFLT, IFEQ и IF_ICMPEQ. Первые две выталкивают верхнее слово из стека и совершают переход в том случае, если это слово меньше 0 или равно 0 соответственно. Команда IF_ICMPEQ берет два верхних слова из стека и совершает переход, если они равны. Во всех трех случаях необходимо считывать новое верхнее слово стека и помещать его в регистр TOS.

Эти три команды сходны. Сначала операнд или операнды помещаются в регистры, затем в TOS записывается новое верхнее слово стека, и, наконец, происходит сравнение и осуществляется переход. Сначала рассмотрим IFLT. Слово, которое нужно проверить, уже находится в регистре TOS, но поскольку команда IFLT выталкивает слово из стека, нужно считать из памяти новую вершину стека и сохранить ее в регистре TOS. Процесс считывания начинается в микрокоманде iflt1. Во время iflt2 слово, которое нужно проверить, сохраняется в регистре OPC, поэтому новое значение можно сразу помесить в регистр TOS, и при этом предыдущее значение не пропадет. В цикле iflt3 новое верхнее слово стека, которое уже находится в MDR, копируется в регистр TOS. Наконец, в цикле iflt4 проверяемое слово (в данный момент оно находится в регистре OPC) пропускается через АЛУ без сохранения результата, после чего проверяется бит N. Если после проверки условие подтверждается, микрокоманда осуществляет переход к T, а если не подтверждается — то к F.

Если условие подтверждается, то происходят те же действия, что и в начале команды GOTO, и далее осуществляется переход к goto2. Если условие не подтверждается, необходима короткая последовательность микрокоманд (F, F2 и F3), чтобы пропустить оставшуюся часть команды (смещение), возвратиться к Main1 и перейти к следующей команде.

Команда IFEQ аналогична команде IFLT, только вместо бита N используется бит Z. В обоих случаях ассемблер должен убедиться, что адреса микрокоманд F и T различаются только крайним левым битом.

Команда IF_ICMPEQ в целом сходна с командой IFLT, только здесь нужно считывать еще и второй операнд. Второй операнд сохраняется в регистре H во время цикла if_icmpeq3, где начинается чтение нового верхнего слова стека. Текущее верхнее слово стека сохраняется в OPC, а новое загружается в регистр TOS. Наконец, микрокоманда if_icmpeq6 аналогична ifeq4.

Теперь рассмотрим команды INVOKEVIRTUAL и IRETURN. Как было описано в разделе «Набор команд IJVM», они служат для вызова процедуры и выхода из нее. Команда INVOKEVIRTUAL представляет собой последовательность из 22 микрокоманд. Это самая сложная команда машины IJVM. Последовательность действий при выполнении этой команды показана на рис. 4.10. 16-битное смещение используется для того, чтобы определить адрес процедуры, которую нужно вызвать. Номер адреса процедуры находится в наборе констант. Следует помнить, что первые 4 байта каждой процедуры — не команды. Это два 16-битных указателя. Первый из них выдает число параметров (включая OBJREF — см. рис. 4.10), а второй — размер области локальных переменных (в словах). Эти поля вызываются через 8-битный порт и объединяются таким же образом, как 16-битное смещение в одной команде.

Затем требуется специальная информация для восстановления предыдущего состояния машины — адрес начала прежней области локальных переменных и старое значение регистра PC. Они сохранены непосредственно над областью локальных переменных под новым стеком. Наконец, вызывается следующий код операции, значение регистра PC увеличивается, происходит переход к циклу Main1 и начинается выполнение следующей команды.

IRETURN — простая команда без операндов. Эта команда просто обращается к первому слову области локальных переменных, чтобы извлечь информацию для возвращения к прежнему состоянию. Затем она восстанавливает предыдущие значения регистров SP, LV и PC и копирует результат выполнения процедуры из нового стека в предыдущий стек, как показано на рис. 4.11.

Разработка микроархитектурного уровня

При разработке микроархитектурного уровня (как и при разработке других уровней) постоянно приходится идти на компромисс. У компьютера есть много важных характеристик: скорость, цена, надежность, простота использования, количество потребляемой энергии, физические размеры. При разработке центрального процессора очень важную роль играет выбор между высокой скоростью и низкой стоимостью. В этом разделе мы подробно рассмотрим данную проблему, покажем преимущества и недостатки каждого из вариантов, а также узнаем, какой производительности можно достичь, какова при этом будет стоимость компьютера и насколько сложным будет аппаратное обеспечение.

Скорость и стоимость

С развитием технологий скорость работы компьютеров стремительно растет. Существует три основных подхода, которые позволяют увеличить скорость выполнения операций:

1. Сокращение количеств циклов, необходимых для выполнения команды.
2. Упрощение организации машины таким образом, чтобы можно было сделать цикл короче.
3. Выполнение нескольких операций одновременно.

Первые два подхода очевидны, но существует огромное количество различных вариантов разработки, которые могут очень сильно повлиять на число циклов, период или (что бывает чаще всего) и то и другое вместе. В этом разделе мы приведем пример того, как кодирование и декодирование операции могут подействовать на цикл.

Число циклов, необходимых для выполнения набора операций, называется **длиной пути**. Иногда длину пути можно уменьшить с помощью дополнительного аппаратного обеспечения. Например, если к регистру PC добавить инкрементор (по сути, это сумматор, у которого один из входов постоянно связан с единицей), то нам больше не придется использовать для этого АЛУ, и следовательно, количество циклов сократится. Однако такой подход не настолько эффективен, как хотелось бы. Часто в том же цикле, в котором значение PC увеличивается на 1, происходит еще и операция чтения, и следующая команда в любом случае не может начаться раньше, поскольку она зависит от данных, которые должны поступить из памяти.

Для сокращения числа циклов, необходимых для вызова команды, требуется нечто большее, чем простое добавление схемы, которая увеличивает PC на 1. Чтобы повысить скорость вызова команды, нужно применить третью технологию — параллельное выполнение команд. Весьма эффективно отделение схем для вызова команд (8-битного порта памяти и регистров PC и MBR), если этот блок сделать функционально независимым от основного тракта данных. Таким образом, он может сам вызывать следующий код операции или операнд. Возможно, он даже будет работать асинхронно относительно другой части процессора и вызывать одну или несколько команд заранее.

Один из наиболее трудоемких процессов при выполнении команд — вызов двубайтного смещения и сохранение его в регистре H для подготовки к сложению (например, при переходе к PC±n байтов). Одно из возможных решений — увеличить порт памяти до 16 битов, но это сильно усложняет операцию, поскольку требуемые 16 битов могут перекрывать границы слова, поэтому даже считывание из памяти 32 битов за один раз не обязательно приведет к вызову обоих нужных нам байтов.

Одновременное выполнение нескольких операций — самый продуктивный подход. Он дает возможность очень сильно повысить скорость работы компьютера. Даже простое перекрытие вызова и выполнения команды чрезвычайно эффективно. При более сложных технологиях допустимо одновременное выполнение нескольких команд. Вообще говоря, эта идея является основой проектов современных компьютеров. Ниже мы обсудим некоторые технические приемы, позволяющие воплотить этот подход.

На одной чаше весов находится скорость, на другой — стоимость. Стоимость можно измерять различными способами, но точное определение стоимости дать очень трудно. В те времена, когда процессоры конструировались из дискретных компонентов, достаточно было подсчитать общее число этих компонентов. В настоящее время процессор целиком помещается на одну микросхему, но большие и более сложные микросхемы стоят гораздо дороже, чем более простые микросхемы небольшого размера. Можно подсчитать отдельные компоненты (транзисторы, вентили, функциональные блоки), но обычно это число не так важно, как размер контактного участка, необходимого для интегральной схемы. Чем больше участок, тем

больше микросхема. И стоимость микросхемы растет гораздо быстрее, чем занимаемое ею пространство. По этой причине разработчики часто измеряют стоимость в единицах, применимых к «недвижимости», то есть с точки зрения пространства, которое требуется для микросхемы (предполагаем, что площадь поверхности измеряется в пикоакрах).

В истории компьютерной промышленности одной из наиболее тщательно проработанных микросхем является двоичный сумматор. Были реализованы тысячи проектов, и самые быстрые двоичные сумматоры очень сильно превышают по скорости самые медленные. Естественно, высокоскоростные сумматоры гораздо сложнее низкоскоростных. Специалистам по разработке систем приходится выбирать определенное соотношение скорости и занимаемого пространства.

Сумматор — не единственный компонент, допускающий различные варианты разработки. Практически любой компонент системы может быть спроектирован таким образом, что он будет функционировать с более высокой или с более низкой скоростью, при этом, естественно, стоимость разных моделей будет различаться. Главной задачей разработчика является определение тех компонентов системы, усовершенствование которых может максимально повлиять на скорость работы компьютера. Интересно отметить, что если какой-нибудь компонент заменить более быстрым, это не обязательно повлечет за собой повышение общей производительности. В следующих разделах мы рассмотрим некоторые вопросы разработки и возможные соотношения цены и скорости.

Одним из ключевых факторов в определении скорости работы генератора синхронизирующего сигнала является количество действий, которые должны быть сделаны за один цикл. Очевидно, чем больше действий должно быть сделано, тем длиннее цикл. Однако все не так просто, ведь аппаратное обеспечение способно выполнять некоторые операции параллельно, поэтому в действительности длина цикла зависит от количества *последовательных* операций в одном цикле.

Должен также учитываться объем выполняемого декодирования. Посмотрите на рис. 4.5. Вспомните, что в АЛУ может передаваться значение одного из девяти регистров, и чтобы определить, какой именно регистр нужно выбрать, требуется всего 4 бита в микрокоманде. К сожалению, такая экономия дорого обходится. Схема декодера вносит дополнительную задержку в работу компьютера. Это значит, что какой бы регистр мы не выбрали, он получит команду немного позже и передаст свое содержимое на шину B немного позже. Следовательно, АЛУ получает входные сигналы и выдает результат также с небольшой задержкой. Соответственно, этот результат поступает на шину C для записи в один из регистров тоже немного позже. Поскольку эта задержка часто является фактором, который определяет длину цикла, это значит, что генератор синхронизирующего сигнала не может функционировать с такой скоростью и весь компьютер должен работать немного медленнее. Таким образом, существует определенная зависимость между скоростью и ценой. Если сократить каждое слово управляющей памяти на 5 битов, это приведет к снижению скорости работы генератора. Инженер при разработке компьютера должен принимать во внимание его предназначение, чтобы сделать правильный выбор. В компьютере с высокой производительностью использовать декодер не рекомендуется, а вот для дешевой машины он вполне подойдет.

Сокращение длины пути

Микроархитектура Mic-1 имеет относительно простую структуру и работает довольно быстро, хотя эти две характеристики очень трудно совместить. В общем случае простые машины не являются высокоскоростными, а высокоскоростные машины довольно сложны. Процессор Mic-1 использует минимум аппаратного обеспечения: 10 регистров, простое АЛУ (см. рис. 3.18), продублированное 32 раза, декодер, схему сдвига, управляющую память и некоторые связующие элементы. Для построения всей системы требуется менее 5000 транзисторов, управляющая память (ПЗУ) и основная память (ОЗУ).

Мы уже показали, как можно воплотить IJVM с помощью микропрограммы, используя небольшое количество аппаратного обеспечения. Теперь рассмотрим альтернативные варианты. Сначала мы изучим, какими способами можно снизить количество микрокоманд в одной команде (то есть каким образом можно сократить длину пути), а затем перейдем к другим подходам.

Слияние цикла интерпретатора с микропрограммой

В микроархитектуре Mic-1 основной цикл состоит из микрокоманды, которая должна выполняться в начале каждой команды IJVM. В некоторых случаях возможно ее перекрытие предыдущей командой. В каком-то смысле эта идея уже получила свое воплощение. Вспомните, что во время цикла `Main1` код следующей операции уже находится в регистре MBR. Этот код операции был вызван или во зремя предыдущего основного цикла (если у предыдущей команды не было операндов), или во время выполнения предыдущей команды.

Эту идею можно развивать и дальше. В некоторых случаях основной цикл можно свести к нулю. Это происходит следующим образом. Рассмотрим каждую последовательность микрокоманд, которая завершается переходом к `Main1`. Каждый раз основной цикл может добавляться в конце этой последовательности (а не в начале следующей), при этом межуровневый переход дублируется много раз (но всегда с одним и тем же набором целевых объектов). В некоторых случаях микрокоманда Mic-1 может сливаться с предыдущими микрокомандами, поскольку эти команды не всегда полностью используются.

В табл. 4.5 приведена последовательность микрокоманд для команды `POP`. Основной цикл идет перед каждой командой и после каждой команды, в таблице этот цикл показан только после команды `POP`. Отметим, что выполнение этой команды занимает 4 цикла: три цикла специальных микрокоманд для команды `POP` и один основной цикл.

Таблица 4.5. Новая микропрограмма для выполнения команды POP

Микрокоманда	Операции	Комментарий
pop1	MAR=SP=SP–1; rd	Считывание второго сверху слова в стеке
pop2		Ожидание, пока из памяти считается новое значение TOS
pop3	TOS=MDR; goto Main1	Копирование нового слова в регистр TOS
Main1	PC=PC+1; fetch; goto(MBR)	Регистр MBR содержит код операции; вызов следующего байта; переход

В табл. 4.6 последовательность сокращена до трех команд за счет того, что в цикле pop2 АЛУ не используется. Отметим, что в конце этой последовательности сразу осуществляется переход к коду следующей команды, поэтому требуется всего 3 цикла. Этот небольшой трюк позволяет сократить время выполнения следующей микрокоманды на один цикл, поэтому, например, последующая команда IADD сокращается с четырех циклов до трех. Это эквивалентно повышению частоты синхронизирующего сигнала с 250 МГц (каждая микрокоманда по 4 нс) до 333 МГц (каждая микрокоманда по 3 нс).

Таблица 4.6. Усовершенствованная микропрограмма для выполнения команды POP

Микрокоманда	Операции	Комментарий
pop1	MAR=SP=SP–1; rd	Считывание второго сверху слова в стеке
Main1.pop	PC=PC+1; fetch	Регистр MBR содержит код операции; вызов следующего байта
pop3	TOS=MDR; goto(MBR)	Копирование нового слова в регистр TOS; переход к коду операции

Команда POP очень хорошо подходит для такой переработки, поскольку она содержит цикл, в котором АЛУ не используется, а основной цикл требует АЛУ. Таким образом, чтобы сократить длину команды на одну микрокоманду, нужно в этой команде найти цикл, где АЛУ не используется. Такие циклы встречаются нечасто, но все-таки встречаются, поэтому установка цикла Main1 в конце каждой последовательности микрокоманд вполне целесообразна. Для этого требуется всего лишь небольшая управляющая память. Итак, мы получили первый способ сокращения длины пути:

помещение основного цикла в конце каждой последовательности микрокоманд.

Трехшинная архитектура

Что еще можно сделать, чтобы сократить длину пути? Можно подвести к АЛУ две полные входные шины, A и B, и следовательно, всего получится три шины. Все (или, по крайней мере, большинство регистров) должны иметь доступ к обеим входным шинам. Преимущество такой системы состоит в том, что есть возможность складывать любой регистр с любым другим регистром за один цикл. Чтобы увидеть, насколько продуктивен такой подход, рассмотрим реализацию команды ILOAD (табл. 4.7).

Таблица 4.7. Микропрограмма для выполнения команды ILOAD

Микрокоманда	Операции	Комментарий
iload1	H=LV	MBR содержит индекс; копирование LV в H
iload2	MAR=MBRU+H; rd	MAR= адрес локальной переменной, которую нужно поместить в стек
iload3	MAR=SP=SP+1	Регистр SP указывает на новую вершину стека; подготовка к записи
iload4	PC=PC+1; fetch; wr	Увеличение PC на 1; вызов следующего кода операции; запись вершины стека
iload5	TOS=MDR; goto Main1	Обновление TOS
Main1	PC=PC+1; fetch; goto(MBR)	Регистр MBR содержит код операции; вызов следующего байта; переход

Мы видим, что в микрокоманде `iload1` значение LV копируется в регистр H. Это нужно только для того, чтобы сложить H с MBRU в микрокоманде `iload2`. В разработке с двумя шинами нет возможности складывать два произвольных регистра, поэтому один из них сначала нужно копировать в регистр H. В трехшинной архитектуре мы можем сэкономить один цикл, как показано в табл. 4.8. Мы добавили основной цикл к команде `ILOAD`, но при этом длина пути не увеличилась и не уменьшилась. Однако дополнительная шина сокращает общее время выполнения команды с шести циклов до пяти циклов. Теперь мы знаем второй способ сокращения длины пути:

переход от двухшинной к трехшинной архитектуре.

Таблица 4.8. Микропрограмма для выполнения команды ILOAD при наличии трехшинной архитектуры

Микрокоманда	Операции	Комментарий
Iload1	MAR=MBRU+LV; rd	MAR= адрес локальной переменной, которую нужно поместить в стек
Iload2	MAR=SP=SP+1	Регистр SP указывает на новую вершину стека; подготовка к записи
Iload3	PC=PC+1; fetch; wr	Увеличение PC на 1; вызов следующего кода операции; запись вершины стека
Iload4	TOS=MDR	Обновление TOS
Iload5	PC=PC+1; fetch; goto(MBR)	Регистр MBR уже содержит код операции; вызов индексного байта

Блок выборки команд

Оба эти способа стоит использовать, но чтобы достичь существенного продвижения, требуется нечто более радикальное. Давайте вернемся чуть-чуть назад и рассмотрим обычные составные части любой команды: поле вызова и поле декодирования. Отметим, что в каждой команде могут происходить следующие операции:

1. Значение PC пропускается через АЛУ и увеличивается на 1.
2. PC используется для вызова следующего байта в потоке команд.
3. Операнды считываются из памяти.
4. Операнды записываются в память.
5. АЛУ выполняет вычисление, и результаты сохраняются в памяти.

Если команда содержит дополнительные поля (для операндов), каждое поле должно вызываться эксплицитно по одному байту. Поле можно использовать только после того, как эти байты будут объединены. При вызове и компоновке поля АЛУ должно для каждого байта увеличивать PC на единицу, а затем объединять получившийся индекс или смещение. Когда помимо выполнения основной работы команды приходится вызывать и объединять поля этой команды, АЛУ используется практически в каждом цикле.

Чтобы объединить основной цикл с какой-нибудь микрокомандой, нужно освободить АЛУ от некоторых таких задач. Для этого можно ввести второе АЛУ, хотя работа полного АЛУ в большинстве случаев не потребуется. Отметим, что

АЛУ часто применяется для копирования значения из одного регистра в другой. Эти циклы можно убрать, если ввести дополнительные тракты данных, которые не проходят через АЛУ. Полезно будет, например, создать тракт от TOS к MDR или от MDR к TOS, поскольку верхнее слово стека часто копируется из одного регистра в другой.

В микроархитектуре Mic-1 с АЛУ можно снять большую часть нагрузки, если создать независимый блок для вызова и обработки команд. Этот блок, который называется **блоком выборки команд**, может независимо от АЛУ увеличивать значение PC на 1 и вызывать байты из потока байтов до того, как они понадобятся. Этот блок содержит инкрементор, который по строению гораздо проще, чем полный сумматор. Разовьем эту идею. Блок выборки команд может также объединять 8-битные и 16-битные операнды, чтобы они могли использоваться сразу, как только они стали нужны. Это можно осуществить, по крайней мере, двумя способами:

1. Блок выборки команд может интерпретировать каждый код операции, определять, сколько дополнительных полей нужно вызвать, и собирать их в регистр, который будет использоваться основным операционным блоком.
2. Блок выборки команд может постоянно предоставлять следующие 8- или 16-битные куски информации независимо от того, имеет это смысл или нет. Тогда основной операционный блок может запрашивать любые данные, которые ему требуются.

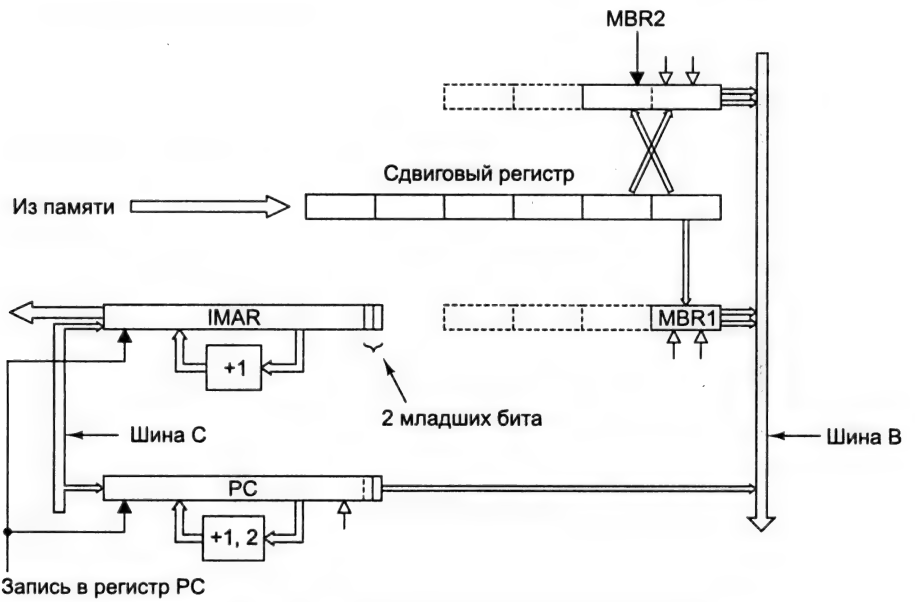


Рис. 4.18. Блок выборки команд для микроархитектуры Mic-1

На рис. 4.18 показан второй способ реализации. Вместо одного 8-разрядного регистра MBR (буферного регистра памяти) здесь есть два регистра MBR: 8-разрядный MBR1 и 16-разрядный MBR2. Блок выборки команд следит за самым последним байтом или байтами, которые поступили в основной операционный блок.

Кроме того, он передает следующий байт в регистр MBR, как и в архитектуре Mic-1, только в данном случае он автоматически определяет, когда значение регистра считано, вызывает следующий байт и сразу загружает его в регистр MBR1. Как и в микроархитектуре Mic-1, он имеет два интерфейса с шиной B: MBR1 и MBR1U. Первый получает знаковое расширение до 32 битов, второй дополнен нулями.

Регистр MBR2 функционирует точно так же, но содержит следующие 2 байта. Он имеет два интерфейса с шиной B: MBR2 и MBR2U, первый из которых расширен по знаку, а второй дополнен до 32 битов нулями.

Блок выборки команд отвечает за вызов байтов. Для этого он использует стандартный 4-байтный порт памяти, вызывая полные 4-байтные слова заранее и загружая следующие байты в сдвиговый регистр, который выдает их по одному или по два за раз в том порядке, в котором они вызываются из памяти. Задача сдвигового регистра — сохранить последовательность поступающих байтов для загрузки в регистры MBR1 и MBR2.

MBR1 всегда содержит самый старший байт сдвигового регистра, а MBR2 — 2 старших байта (старший байт — левый байт), которые формируют 16-битное целое число (см. рис. 4.14, *б*). Два байта в регистре MBR2 могут быть из различных слов памяти, поскольку команды IJVM никак не связаны с границами слов.

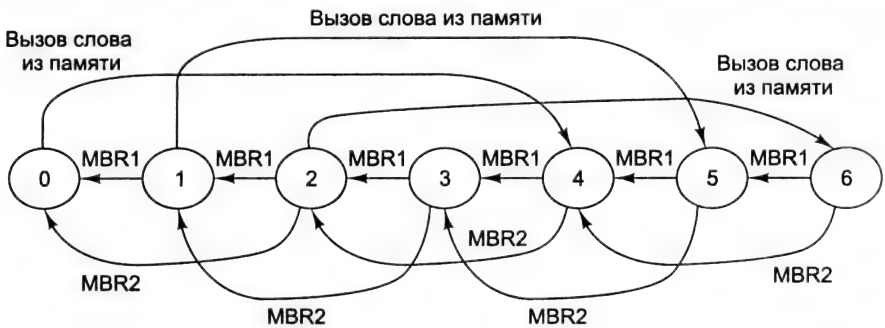
Всякий раз, когда считывается регистр MBR1, значение сдвигового регистра сдвигается вправо на 1 байт. Всякий раз, когда считывается регистр MBR2, значение сдвигового регистра сдвигается вправо на 2 байта. Затем в регистры MBR1 и MBR2 загружается самый старший байт и пара самых старших байтов соответственно. Если в данный момент в сдвиговом регистре осталось достаточно места для целого слова, блок выборки команд начинает цикл обращения к памяти, чтобы считать следующее слово. Мы предполагаем, что когда считывается любой из регистров MBR, он перезагружается к началу следующего цикла, поэтому новое значение можно считывать уже в последующих циклах.

Блок выборки команд может быть смоделирован в виде **автомата с конечным числом состояний** (или **конечного автомата**), как показано на рис. 4.19. Во всех конечных автоматах есть **состояния** (на рисунке это кружочки) и **переходы** (это дуги от одного состояния к другому). Каждое состояние — это одна из возможных ситуаций, в которой может находиться конечный автомат. Данный конечный автомат имеет семь состояний, которые соответствуют семи состояниям сдвигового регистра, показанного на рис. 4.18. Семь состояний соответствуют количеству байтов, которые находятся в данный момент в регистре (от 0 до 6 включительно).

Каждая дуга репрезентирует событие, которое может произойти. В нашем конечном автомате возможны три различных события. Первое событие — чтение одного байта из регистра MBR1. Оно активизирует сдвиговый регистр, самый правый байт в нем убирается, и осуществляется переход в другое состояние (меньшее на 1). Второе событие — чтение двух байтов из регистра MBR2. При этом осуществляется переход в состояние, меньшее на 2 (например, из состояния 2 в состояние 0 или из состояния 5 в состояние 3). Оба эти перехода вызывают перезагрузку регистров MBR1 и MBR2. Когда конечный автомат переходит в состояния 0, 1 или 2, начинается процесс обращения к памяти, чтобы вызвать новое слово (предпо-

лагается, что память уже не занята считыванием слова). При поступлении слова номер состояния увеличивается на 4.

Чтобы функционировать правильно, схема выборки команд должна блокироваться в том случае, если от нее требуют произвести какие-то действия, которые она выполнить не может (например, передать значение в MBR2, когда в сдвиговом регистре находится только 1 байт, а память все еще занята вызовом нового слова). Кроме того, блок выборки команд не может выполнять несколько операций одновременно, поэтому вся поступающая информация должна передаваться последовательно. Наконец, всякий раз, когда изменяется значение PC, блок выборки команд должен обновляться. Все эти детали усложняют работу блока. Однако многие устройства аппаратного обеспечения конструируются в виде конечных автоматов.



События
MBR1: Чтение регистра MBR1
MBR2: Чтение регистра MBR2
Вызов слова из памяти: это событие означает считывание слова из памяти и помещение 4 байтов в сдвиговый регистр

Рис. 4.19. Автомат с конечным числом состояний для реализации блока выборки команд

Блок выборки команд имеет свой собственный регистр адреса ячейки памяти (IMAR), который используется для обращения к памяти, когда нужно вызвать новое слово. У этого регистра есть специальный инкрементор, поэтому основному АЛУ не требуется прибавлять 1 к значению PC для вызова следующего слова. Блок выборки команд должен контролировать шину C, чтобы каждый раз при загрузке регистра PC новое значение PC также копировалось в IMAR. Поскольку новое значение в регистре PC может быть и не на границе слова, блок выборки команд должен вызвать нужное слово и скорректировать значение сдвигового регистра соответствующим образом.

Основной операционный блок записывает значение в PC только в том случае, если необходимо изменить характер последовательности байтов. Это происходит в команде перехода, в команде `INVOKEVIRTUAL` и команде `IRETURN`.

Поскольку микропрограмма больше не увеличивает PC явным образом при вызове кода операции, блок выборки команд должен обновлять PC сам. Как это происходит? Блок выборки команд способен распознать, что байт из потока команд получен, то есть, что значения регистров MBR1 и MBR2 (или их вариантов

без знака) уже считаны. С регистром PC связан отдельный инкрементор, который увеличивает значение на 1 или на 2 в зависимости от того, сколько байтов получено. Таким образом, регистр PC всегда содержит адрес первого еще не полученного байта. В начале каждой команды в регистре MBR находится адрес кода операции этой команды.

Отметим, что существует два разных инкрементора, которые выполняют разные функции. Регистр PC считает *байты* и увеличивает значение на 1 или на 2. Регистр IMAR считает *слова* и увеличивает значение только на 1 (для 4 новых байтов). Как и MAR, регистр IMAR соединен с адресной шиной, при этом бит 0 регистра IMAR связан с адресной линией 2 и т. д. (перекос шины), чтобы осуществлять переход от адреса слова к адресу байта.

Мы скоро увидим, что если нет необходимости увеличивать значение PC в основном цикле, это приводит к большому выигрышу, поскольку обычно в микрокоманде, в которой происходит увеличение PC, кроме этого больше ничего не происходит. Если эту команду устранить, длина пути сократится. Однако для увеличения скорости работы машины требуется больше аппаратного обеспечения. Таким образом, мы пришли к третьему способу сокращения длины пути:

команды из памяти должны вызываться специализированным функциональным блоком.

Микроархитектура с упреждающей выборкой команд из памяти: Mic-2

Блок выборки команд может сильно сократить длину пути средней команды. Во-первых, он полностью устраняет основной цикл, поскольку в конце каждой команды просто стразу осуществляется переход к следующей команде. Во-вторых, АЛУ не нужно увеличивать значение PC. В-третьих, блок выборки команд сокращает длину пути всякий раз, когда вычисляется 16-битный индекс или смещение, поскольку он объединяет 16-битное значение и сразу передает его в АЛУ в виде 32-битного значения, избегая необходимости производить объединение в регистре H. На рис. 4.20 показана микроархитектура Mic-2, которая представляет собой усовершенствованную версию Mic-1, к которой добавлен блок выборки команд, изображенный на рис. 4.18. Микропрограмма для усовершенствованной машины приведена в табл. 4.9.

Чтобы продемонстрировать, как работает Mic-2, рассмотрим команду `IADD`. Она берет второе слово из стека и выполняет сложение, как и раньше, но только сейчас ей не нужно осуществлять переход к `Main1` после завершения операции, чтобы увеличить значение PC и перейти к следующей микрокоманде. Когда блок выборки команд распознает, что в цикле `iadd3` произошло обращение к регистру MBR1, его внутренний сдвиговый регистр сдвигает все вправо и перезагружает MBR1 и MBR2. Он также осуществляет переход в состояние, которое на 1 ниже текущего. Если новое состояние — это состояние 2, блок выборки команд начинает вызов слова из памяти. Все это происходит в аппаратном обеспечении. Микропрограмма ничего не должна делать. Именно поэтому команду `IADD` можно сократить с пяти до трех микрокоманд.

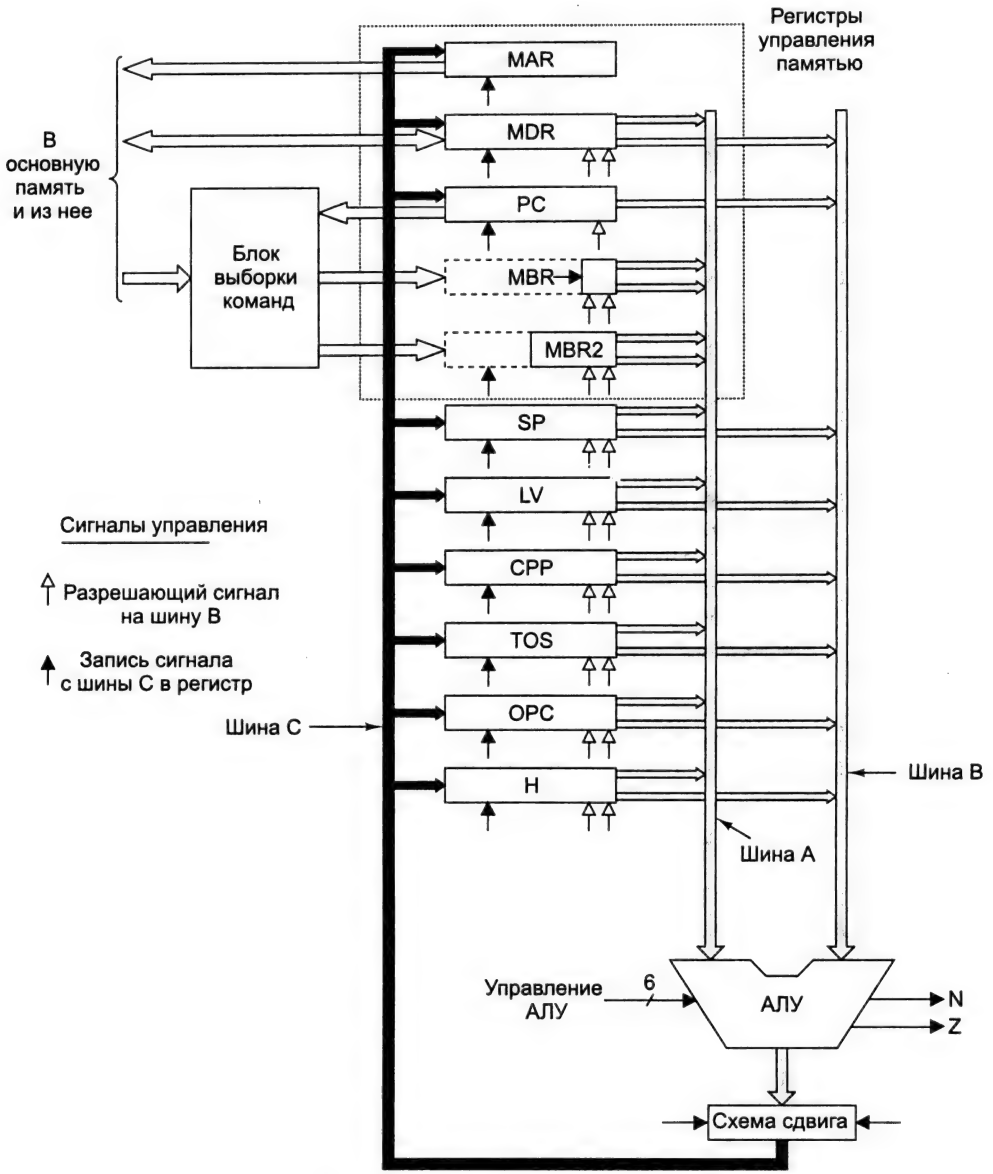


Рис. 4.20. Тракт данных для Mic-2

Таблица 4.9. Микропрограмма для Mic-2

Микрокоманда	Операции	Комментарий
nop1	goto (MBR)	Переход к следующей команде
iadd1	MAR=SP=SP–1; rd	Чтение слова, идущего после верхнего слова стека
iadd2	H=TOS	H = вершина стека

продолжение

Таблица 4.9 (*продолжение*)

Микрокоманда	Операции	Комментарий
iadd3	MDR=TOS=MDR+H; wr;goto (MBR1)	Суммирование двух верхних слов; запись суммы в верхнюю позицию стека
isub1	MAR=SP=SP–1; rd	Чтение слова, идущего после верхнего слова стека
isub2	H=TOS	H = вершина стека
isub3	MDR=TOS=MDR–H; wr; goto (MBR1)	Вычитание TOS из вызванного значения TOS-1
iand1	MAR=SP=SP–1; rd	Чтение слова, идущего после верхнего слова стека
iand2	H=TOS	H = вершина стека
iand3	MDR=TOS=MDR И H; wr; goto (MBR1)	Логическое умножение вызванного значения TOS-1 и TOS (операция И)
ior1	MAR=SP=SP–1; rd	Чтение слова, идущего после верхнего слова стека
ior2	H=TOS	H = вершина стека
ior3	MDR=TOS=MDR ИЛИ H; wr; goto (MBR1)	Логическое сложение вызванного значения TOS-1 и TOS (операция ИЛИ)
dup1	MAR=SP=SP+1	Увеличение SP на 1 и копирование результата в регистр MAR
dup2	MDR=TOS; wr; goto (MBR1)	Запись нового слова в стек
pop1	MAR=SP=SP–1; rd	Чтение слова, идущего после верхнего слова стека
pop2		Программа ждет, пока закончится процесс чтения
pop3	TOS=MDR; goto (MBR1)	Копирование нового слова в регистр TOS
swap1	MAR=SP–1; rd	Чтение второго слова из стека; установка регистра MAR на значение SP
swap2	MAR=SP	Подготовка к записи нового второго слова стека
swap3	H=MDR; wr	Сохранение нового значения TOS; запись второго слова стека
swap4	MDR=TOS	Копирование прежнего значения TOS в регистр MDR
swap5	MAR=SP–1; wr	Запись прежнего значения TOS на второе место в стеке
swap6	TOS=H; goto (MBR1)	Обновление TOS
bipush1	SP=MAR=SP+1	Установка регистра MAR для записи в новую вершину стека
bipush2	MDR=TOS=MBR1; wr; goto (MBR1)	Обновление стека в регистре TOS и памяти
iload1	MAR=LV+MBR1U; rd	Перемещение значения LV с индексом в регистр MAR; чтение операнда
iload2	MAR=SP=SP+1	Увеличение SP на 1; перемещение нового значения SP в регистр MAR
iload3	TOS=MDR; wr; goto (MBR1)	Обновление стека в регистре TOS и памяти
istore1	MAR=LV+MBR1U	Установка регистра MAR на значение LV+индекс
istore2	MDR=TOS; wr	Копирование значения TOS для сохранения в памяти
istore3	MAR=SP=SP–1; rd	Уменьшение SP на 1; чтение нового значения TOS
istore4		Машина ждет окончания процесса чтения
istore5	TOS=MDR; goto (MBR1)	Обновление TOS

Микрокоманда	Операции	Комментарий
wide1	goto(MBR1 ИЛИ 0x100)	Следующий адрес — 0x100 ИЛИ код операции
wide_iload1	MAR=LV+MBR2U; rd; goto iload2	То же, что iload1, но с использованием 2-байтного индекса
wide_istore1	MAR=LV+MBR2U; goto istore2	То же, что istore1, но с использованием 2-байтного индекса
ldc_w1	MAR=CPP+MBR2U; rd; goto iload2	То же, что wide_iload1, но индексирование осуществляется из регистра CPP
iinc1	MAR=LV+MBR1U; rd	Установка регистра MAR на значение LV+индекс; чтение этого значения
iinc2	H=MBR1	Присваивание регистру H константы
iinc3	MDR=MDR+H; wr; goto (MBR1)	Увеличение на константу и обновление
goto1	H=PC–1	Копирование значения PC в регистр H
goto2	PC=H+MBR2	Прибавление смещения и обновление PC
goto3		Машина ждет, пока блок выборки команд вызовет новый код операции
goto4	goto (MBR1)	Переход к следующей команде
iflt1	MAR=SP=SP–1; rd	Чтение второго сверху слова в стеке
iflt2	OPC=TOS	Временное сохранение TOS в OPC
iflt3	TOS=MDR	Запись новой вершины стека в TOS
iflt4	N=OPC; if(N) goto T; else goto F	Переход в бит N
ifeq1	MAR=SP=SP–1; rd	Чтение второго сверху слова в стеке
ifeq2	OPC=TOS	Временное сохранение TOS в OPC
ifeq3	TOS=MDR	Запись новой вершины стека в TOS
ifeq4	Z=OPC; if(Z) goto T; else goto F	Переход в бит Z
if_icmpeq1	MAR=SP=SP–1; rd	Чтение второго сверху слова в стеке
if_icmpeq2	MAR=SP=SP–1	Установка регистра MAR на чтение новой вершины стека
if_icmpeq3	H=MDR; rd	Копирование второго слова из стека в регистр H
if_icmpeq4	OPC=TOS	Временное сохранение TOS в OPC
if_icmpeq5	TOS=MDR	Помещение новой вершины стека в TOS
if_icmpeq6	Z=H–OPC; if(Z) goto T; else goto F	Если два верхних слова равны, осуществляется переход к T; если они не равны, осуществляется переход к F
T	H=PC–1; goto goto2	То же, что goto1
F	H=MBR2	Игнорирование байтов, находящихся в регистре MBR2
F2	goto (MBR1)	
invoke_virtual1	MAR=CPP+MBR2U; rd	Помещение адреса указателя процедуры в регистр MAR
invoke_virtual2	OPC=PC	Сохранение значения PC в регистре OPC
invoke_virtual3	PC=MDR	Установка PC на первый байт кода процедуры
invoke_virtual4	TOS=SP–MBR2U	TOS = адрес OBJREF–1
invoke_virtual5	TOS=MAR=H=TOS+1	TOS = адрес OBJREF

продолжение

Таблица 4.9 (*продолжение*)

Микрокоманда	Операции	Комментарий
invoke_virtual6	MDR=SP+MBR2U+1; wr	Перезапись OBJREF со связующим указателем
invoke_virtual7	MAR=SP=MDR	Установка регистров SP и MAR на адрес ячейки, в которой содержится старое значение PC
invoke_virtual8	MDR=OPC; wr	Подготовка к сохранению старого значения PC
invoke_virtual9	MAR=SP=SP+1	Увеличение SP на 1; теперь SP указывает на ячейку, в которой хранится старое значение LV
invoke_virtual10	MDR=LV; wr	Сохранение старого значения LV
invoke_virtual11	LV=TOS; goto (MBR1)	Установка значения LV на нулевой параметр
ireturn1	MAR=SP=LV; rd	Переустановка регистров SP и MAR для чтения связующего указателя
ireturn2		Процесс считывания связующего указателя
ireturn3	LV=MAR=MDR; rd	Установка регистров LV и MAR на связующий указатель; чтение старого значения PC
ireturn4	MAR=LV+1	Установка регистра MAR на старое значение LV; чтение старого значения LV
ireturn5	PC=MDR; rd	Восстановление PC
ireturn6	MAR=SP	
ireturn7	LV=MDR	Восстановление LV
ireturn8	MDR=TOS; wr; goto (MBR1)	Сохранение результата в изначальной вершине стека

Микроархитектура Mic-2 совершенствует некоторые команды в большей степени, чем другие. Команда `LDC_W` сокращается с 9 до 3 микрокоманд, и следовательно, время выполнение команды уменьшается в три раза. Несколько по-другому дело обстоит с командой `SWAP`: изначально там было 8 команд, а стало 6. Для общей производительности компьютера играет роль сокращение наиболее часто повторяющихся команд. Это команды `ILOAD` (было 6, сейчас 3), `IADD` (было 4, сейчас 3) и `IF_ICMPEQ` (было 13, сейчас 10 для случая, если слова равны; было 10, сейчас 8 для случая, если слова не равны). Чтобы вычислить, насколько улучшилась производительность, можно проверить эффективность системы по эталонному тесту, но и без этого ясно, что здесь наблюдается огромный выигрыш в скорости.

Конвейерная архитектура: Mic-3

Ясно, что Mic-2 — это усовершенствованная микроархитектура Mic-1. Она работает быстрее и использует меньше управляющей памяти, хотя стоимость блока выборки команд, несомненно, превышает ту сумму, которая выигрывается за счет сокращения пространства при уменьшении управляющей памяти. Таким образом, машина Mic-2 работает значительно быстрее при минимальном росте стоимости. Давайте посмотрим, можно ли еще больше повысить скорость.

А что, если попробовать уменьшить время цикла? В значительной степени время цикла определяется базовой технологией. Чем меньше транзисторы и чем меньше физическое расстояние между ними, тем быстрее может работать задающий генератор. В технологии, которую мы рассматриваем, время, затрачиваемое на

прохождение через тракт данных, фиксировано (по крайней мере, с нашей точки зрения). Тем не менее у нас есть некоторая свобода действий, и далее мы используем ее в полной мере.

Еще один вариант усовершенствования — ввести в машину больше параллелизма. В данный момент микроархитектура Mic-2 выполняет большинство операций последовательно. Она помещает значения регистров на шины, ждет, пока АЛУ и схема сдвига обработают их, а затем записывает результаты обратно в регистры. Если не учитывать работу блока выборки команд, никакого параллелизма здесь нет. Внедрение дополнительных средств параллельной обработки дает нам большие возможности.

Как мы уже говорили, длительность цикла определяется временем, необходимым для прохождения сигнала через тракт данных. На рис. 4.2 показано распределение этой задержки между различными компонентами во время каждого цикла. В цикле тракта данных есть три основных компонента:

1. Время, которое требуется на передачу значений выбранных регистров в шины A и B.
2. Время, которое требуется на работу АЛУ и схемы сдвига.
3. Время, которое требуется на передачу полученных значений обратно в регистры и сохранение этих значений.

На рис. 4.21 показана новая трехшинная архитектура с блоком выборки команд и тремя дополнительными защелками (регистрами), каждая из которых расположена в середине каждой шины. Эти регистры записываются в каждом цикле. Они делят тракт данных на отдельные части, которые теперь могут функционировать независимо друг от друга. Мы будем называть такую архитектуру **конвейерной моделью** или **Mic-3**.

Зачем нужны эти три дополнительных регистра? Ведь теперь для прохождения сигнала через тракт данных требуется три цикла: один — для загрузки регистров A и B, второй — для запуска АЛУ и схемы сдвига и загрузки регистра C, и третий — для сохранения значения регистра-защелки C обратно в нужные регистры. Мы что, сумасшедшие? (*Подсказка*: нет). Существует целых две причины введения дополнительных регистров:

1. Мы можем повысить тактовую частоту, поскольку максимальная задержка теперь стала короче.
2. Во время каждого цикла мы можем использовать все части тракта данных.

После разбиения тракта данных на три части максимальная задержка прохождения сигнала уменьшается, и в результате тактовая частота может повыситься. Предположим, что если разбить цикл тракта данных на три примерно равных интервала, тактовая частота увеличится втрое. (На самом деле это не так, поскольку мы добавили в тракт данных еще два регистра, но в качестве первого приближения это допустимо.)

Поскольку мы предполагаем, что все операции чтения из памяти и записи в память выполняются с использованием кэш-памяти первого уровня и эта кэш-память построена из того же материала, что и регистры, то следовательно, операция с памятью занимает один цикл. На практике, однако, этого не так легко достичь.

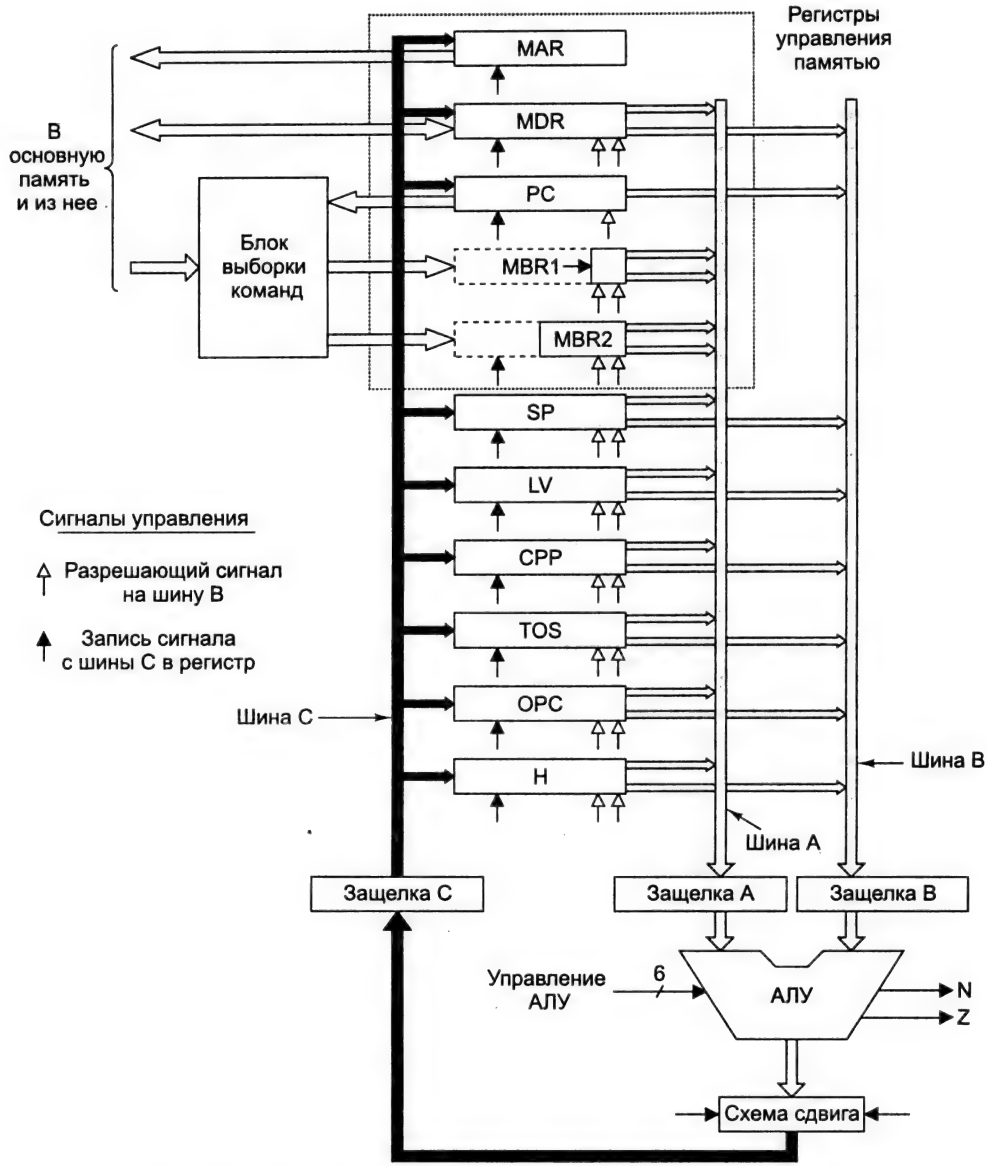


Рис. 4.21. Тракт данных с тремя шинами в микроархитектуре Mic-3

Второй пункт связан с общей производительностью, а не со скоростью выполнения отдельной команды. В микроархитектуре Mic-2 во время первой и третьей части каждого цикла АЛУ простаивает. Если разделить тракт данных на три части, то появляется возможность использовать АЛУ в каждом цикле, вследствие чего производительность машины увеличивается в три раза.

А теперь посмотрим, как работает тракт данных Mic-3. Перед тем как начать, нужно ввести определенные обозначения для защелок. Проще всего назвать за-

щелки A, B и C и считать их регистрами, подразумевая ограничения тракта данных. В таблице 4.10 приведен кусок программы для микроархитектуры Mic-2 (реализация команды SWAP).

Таблица 4.10. Программа Mic-2 для команды SWAP

Микрокоманда	Операции	Комментарий
swap1	MAR=SP–1; rd	Чтение второго слова из стека; установка MAR на SP
swap2	MAR=SP	Подготовка к записи нового второго слова
swap3	H=MDR; wr	Сохранение нового значения TOS; запись второго слова в стек
swap4	MDR=TOS	Копирование старого значения TOS в регистр MDR
swap5	MAR=SP–1; wr	Запись старого значения TOS на второе место в стеке
swap6	TOS=H; goto(MBR1)	Обновление TOS

Давайте перепишем эту последовательность для Mic-3. Следует помнить, что теперь работа тракта данных занимает три цикла: один — для загрузки регистров A и B, второй — для выполнения операции и загрузки регистра C и третий — для записи результатов в регистры. Каждый из этих участков будем называть **микрошагом**.

Реализация команды SWAP для Mic-3 показана в табл. 4.11. В цикле 1 мы начинаем микрокоманду swap1, копируя значение SP в регистр B. Не имеет никакого значения, что происходит в регистре A, поскольку чтобы отнять 1 из B, ENA (сигнал разрешения A) блокируется (см. табл. 4.1). Для простоты мы не показываем присваивания, которые не используются. В цикле 2 мы производим операцию вычитания. В цикле 3 результат сохраняется в регистре MAR, и после этого, в конце третьего цикла, начинается процесс чтения. Поскольку чтение из памяти занимает один цикл, закончится он только в конце четвертого цикла. Это показано присваиванием значения регистру MDR в цикле 4. Значение из MDR можно считывать не раньше пятого цикла.

Таблица 4.11. Реализация команды SWAP для Mic-3

	Swap1	Swap2	Swap3	Swap4	Swap5	Swap6
Цикл	MAR=SP–1;rd	MAR=SP	H=MDR;wr	MDR=TOS	MAR=SP–1;wr	TOS=H; goto(MBR1)
1	B=SP					
2	C=B–1	B=SP				
3	MAR=C; rd	C=B				
4	MDR=Mem	MAR=C				
5			B=MDR			
6			C=B	B=TOS		
7			H=C; wr	C=B	B=SP	
8			Mem=MDR	MDR=C	C=B–1	B=H
9					MAR=C; wr	C=B
10					Mem=MDR	TOS=C
11						goto(MBR1)

А теперь вернемся к циклу 2. Мы можем разбить микрокоманду swap2 на микрошаги и начать их выполнение. В цикле 2 мы копируем значение SP в регистр B, затем пропускаем значение через АЛУ в цикле 3 и, наконец, сохраняем его в регистре MAR в цикле 4. Пока все хорошо. Должно быть ясно, что если мы сможем начинать новую микрокоманду в каждом цикле, скорость работы машины увеличится в три раза. Такое повышение скорости происходит за счет того, что машина Mic-3 производит в три раза больше циклов в секунду, чем Mic-2. Фактически мы построили конвейерный процессор.

К сожалению, мы наткнулись на преграду в цикле 3. Мы бы рады начать микрокоманду swap3, но эта микрокоманда сначала пропускает значение MDR через АЛУ, а значение MDR не будет получено из памяти до начала цикла 5. Ситуация, когда следующий микрошаг не может начаться, потому что перед этим нужно получить результат выполнения предыдущего микрошага, называется **реальной взаимозависимостью** или **RAW-взаимозависимостью (Read After Write — чтение после записи)**. В такой ситуации требуется считать значение регистра, которое еще не записано. Единственное разумное решение в данном случае — отложить начало микрокоманды swap3 до того момента, когда значение MDR станет доступным, то есть до пятого цикла. Ожидание нужного значения называется **простаиванием**. После этого мы можем начинать выполнение микрокоманд в каждом цикле, поскольку таких ситуаций больше не возникает, хотя имеется пограничная ситуация: микрокоманда swap6 считывает значение регистра H в цикле, который следует сразу после записи этого регистра в микрокоманде swap3. Если бы значение этого регистра считывалось в микрокоманде swap5, машине пришлось бы простаивать один цикл.

Хотя программа Mic-3 занимает больше циклов, чем программа Mic-2, она работает гораздо быстрее. Если время цикла микроархитектуры Mic-3 составляет ΔT наносекунд, то для выполнения команды SWAP машине Mic-3 требуется $11\Delta T$ нс, а машине Mic-2 нужно 6 циклов по $3\Delta T$ нс каждый, то есть всего $18\Delta T$ нс. Конвейеризация увеличивает скорость работы компьютера, даже несмотря на то, что один раз приходится простаивать из-за явления взаимозависимости.

Конвейеризация является ключевой технологией во всех современных процессорах, поэтому важно хорошо понимать эту технологию. На рис. 4.22 графически проиллюстрирована конвейеризация тракта данных, изображенного на рис. 4.21. В первой колонке демонстрируется, что происходит во время цикла 1, вторая колонка представляет цикл 2 и т. д. (предполагается, что простаиваний нет). Закрашенная область на рисунке для цикла 1 и команды 1 означает, что блок выборки команд занят вызовом команды 1. В цикле 2 значения регистров, вызванных командой 1, загружаются в A и B, а в это время блок выборки команд занимается вызовом команды 2. Все это также показано с помощью закрашенных серым прямоугольников.

Во время цикла 3 команда 1 использует АЛУ и схему сдвига, регистры A и B загружаются для команды 2, а команда 3 вызывается. Наконец, во время цикла 4 работают все 4 команды одновременно. Сохраняются результаты выполнения команды 1, АЛУ выполняет вычисления для команды 2, регистры A и B загружаются для команды 3, а команда 4 вызывается.

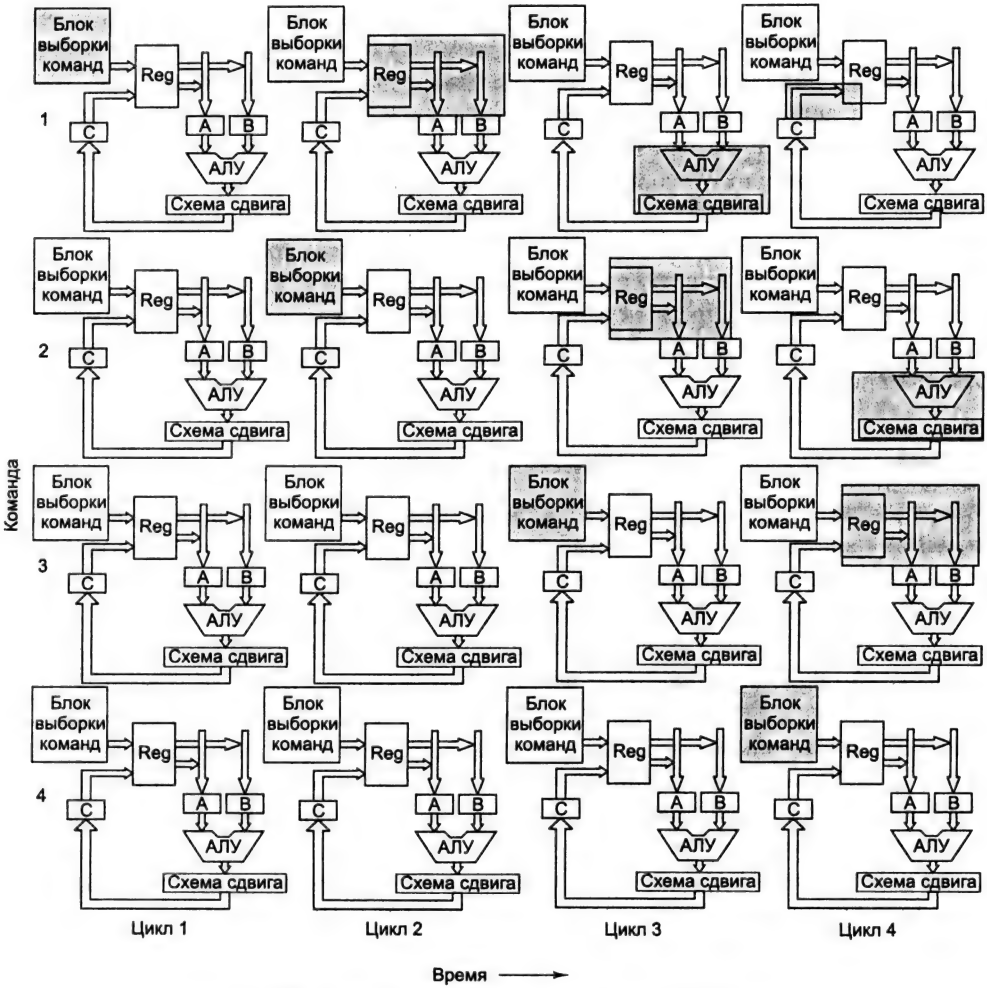


Рис. 4.22. Графическое изображение работы конвейера

Если бы мы показали цикл 5 и следующие, модель была бы точно такой же, как в цикле 4: все четыре части тракта данных работали бы независимо друг от друга. Данный конвейер содержит 4 стадии: для вызова команд, для доступа к операндам, для работы АЛУ и для записи результата обратно в регистры. Он похож на конвейер, изображенный на рис. 2.3, *а*, только у него отсутствует стадия декодирования (расшифровки). Здесь важно подчеркнуть, что хотя выполнение одной команды занимает 4 цикла, в каждом цикле начинается новая команда и завершается предыдущая.

Можно рассматривать схему на рис. 4.22 не вертикально (по колонкам), а горизонтально (по строчкам). При выполнении команды 1 в цикле 1 функционирует блок выборки команд. В цикле 2 значения регистров помещаются на шины A и B. В цикле три происходит работа АЛУ и схемы сдвига. Наконец, в цикле 4 полученные результаты сохраняются в регистрах. Отметим, что имеется 4 доступные части

аппаратного обеспечения, и во время каждого цикла определенная команда использует только одну из них, оставляя свободными другие части для других команд.

Проведем аналогию с конвейером на заводе по производству машин. Чтобы изложить основную суть работы такого конвейера, представим, что ровно каждую минуту звучит гонг, и в этот момент все машины передвигаются по линии на один пункт. В каждом пункте рабочие выполняют определенную операцию с машиной, которая находится перед ними, например ставят колеса или тормоза. При каждом ударе гонга (это 1 цикл) одна новая машина поступает на конвейер и одна собранная машина сходит с конвейера. Завод выпускает одну машину в минуту независимо от того, сколько времени занимает сборка одной машины. В этом и состоит суть работы конвейера. Такой подход в равной степени применим и к процессорам, и к производству машин.

Конвейер с 7 стадиями: Mic-4

Мы не упомянули о том факте, что каждая микрокоманда выбирает следующую за ней микрокоманду. Большинство из них просто выбирают следующую команду в текущей последовательности, но последняя из них, например swap6, часто совершает межуровневый переход, который останавливает работу конвейера, поскольку после этого перехода вызывать команды заранее уже становится невозможно. Поэтому нам нужно придумать лучшую технологию.

Следующая (и последняя) микроархитектура — Mic-4. Ее основные части проиллюстрированы на рис. 4.23, но значительное количество деталей не показано, чтобы сделать схему более понятной. Как и Mic-3, эта микроархитектура содержит блок выборки команд, который заранее вызывает слова из памяти и сохраняет различные значения MBR.

Блок выборки команд передает входящий поток байтов в новый компонент — **блок декодирования**. Этот блок содержит внутреннее ПЗУ, которое индексируется кодом операции IJVM. Каждый элемент (ряд) блока состоит из двух частей: длины команды IJVM и индекса в другом ПЗУ — ПЗУ микроопераций. Длина команды IJVM нужна для того, чтобы блок декодирования мог разделить входящий поток байтов и установить, какие байты являются кодами операций, а какие операндами. Если длина текущей команды составляет 1 байт (например, длина команды `POP`), то блок декодирования определяет, что следующий байт — это код операции. Если длина текущей команды составляет 2 байта, блок декодирования определяет, что следующий байт — это операнд, сразу за которым следует другой код операции. Когда появляется префиксная команда `WIDE`, следующий байт преобразуется в специальный расширенный код операции, например, `WIDE+ILOAD` превращается в `WIDE_ILOAD`.

Блок декодирования передает индекс в ПЗУ микроопераций, который он находит в своей таблице, следующему компоненту, **блоку формирования очереди**. Этот блок содержит логические схемы и две внутренние таблицы: одна — для ПЗУ и одна — для ОЗУ. В ПЗУ находится микропрограмма, причем каждая команда IJVM содержит набор последовательных элементов, которые называются **микрооперации-**

ями. Эти элементы должны быть расположены в строгом порядке, и, например, переход из `wide_iload2` в `iload2`, который допустим в микроархитектуре Mic-2, не разрешается. Каждая последовательность микроопераций должна выполняться полностью, в некоторых случаях последовательности дублируются.

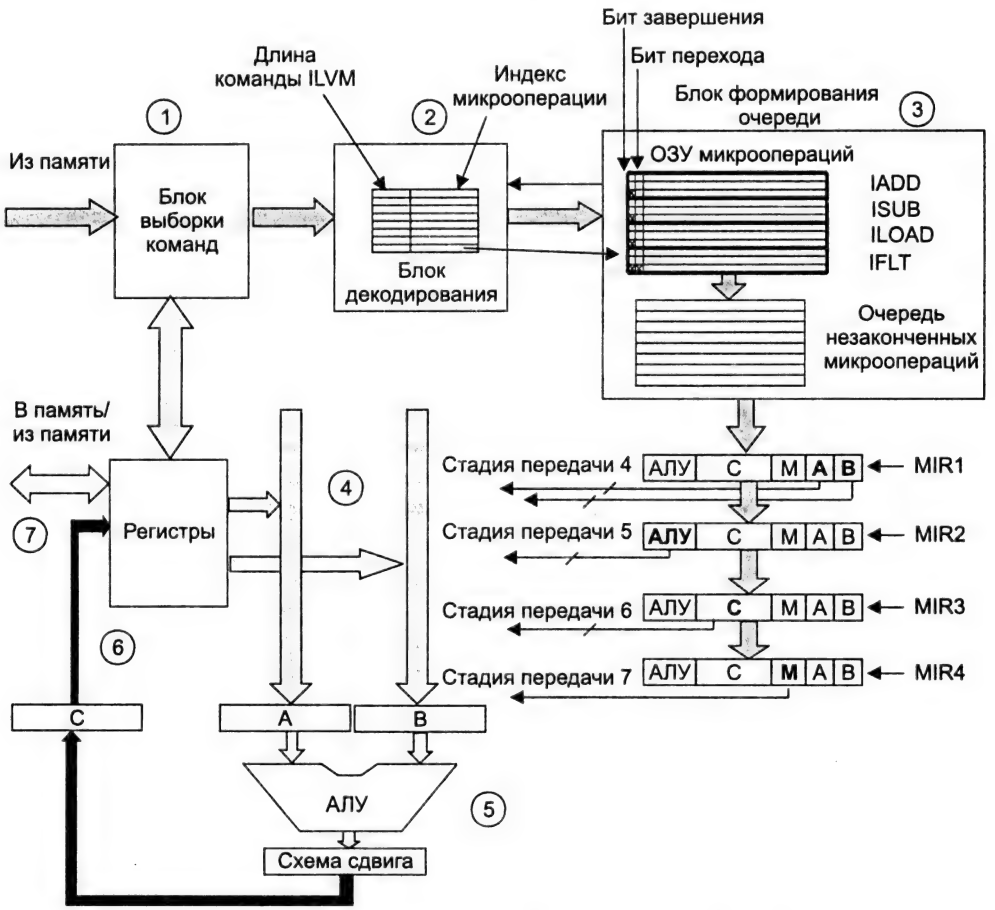


Рис. 4.23. Основные компоненты микроархитектуры Mic-4

Структура микрооперации сходна со структурой микрокоманды (см. рис. 4.4), только в данном случае поля NEXT_ADDRESS и JAM отсутствуют и требуется новое поле для определения входа на шину A. Имеется также два новых бита: бит завершения (Final bit) и бит перехода (Goto bit). Бит завершения устанавливается на последней микрооперации каждой последовательности (чтобы обозначить эту операцию). Бит перехода нужен для указания на микрооперации, которые являются условными микропереходами. По формату они отличаются от обычных микроопераций. Они состоят из битов JAM и индекса в ПЗУ микроопераций. Микрокоманды, которые раньше осуществляли какие-либо действия с трактом данных, а также выполняли условные микропереходы (например, `iflt4`), теперь нужно разбивать на две микрооперации.

Блок формирования очереди работает следующим образом. Он получает от блока декодирования индекс микрооперации ПЗУ. Затем он отыскивает микрооперацию и копирует ее во внутреннюю очередь. Затем он копирует следующую микрооперацию в ту же очередь, а также следующую за этой микрооперацией. Так продолжается до тех пор, пока не появится микрооперация с битом завершения. Тогда блок копирует эту последнюю микрооперацию и останавливается. Если блоку не встретилась микрооперация с битом перехода и у него осталось достаточно свободного пространства, он посылает сигнал подтверждения приема блоку декодирования. Когда блок декодирования воспринимает сигнал подтверждения, он посылает блоку формирования очереди следующую команду IJVM.

Таким образом, последовательность команд IJVM в памяти в конечном итоге превращается в последовательность микроопераций в очереди. Эти микрооперации передаются в регистры MIR, которые посылают сигналы тракту данных. Но есть еще один фактор, который нам нужно рассмотреть: поля каждой микрооперации не действуют одновременно. Поля A и B активны во время первого цикла, поле АЛУ активно во время второго цикла, поле C активно во время третьего цикла, а все операции с памятью происходят в четвертом цикле.

Чтобы все эти операции выполнялись правильно, мы ввели 4 независимых регистра MIR в схему на рис. 4.23. В начале каждого цикла (на рис. 4.2 это время Δw) значение MIR3 копируется в регистр MIR4, значение MIR2 копируется в регистр MIR3, значение MIR1 копируется в регистр MIR2, а в MIR1 загружается новая микрооперация из очереди. Затем каждый регистр MIR выдает сигналы управления, но используются только некоторые из них. Поля A и B из регистра MIR1 применяются для выбора регистров, которые запускают защелки A и B, а поле АЛУ в регистре MIR1 не используется и не связано ни с чем на тракте данных.

В следующем цикле микрооперация передается в регистр MIR2, а выбранные регистры в данный момент находятся в защелках A и B. Поле АЛУ теперь используется для запуска АЛУ. В следующем цикле поле C запишет результаты обратно в регистры. После этого микрооперация передается в регистр MIR4 и инициирует любую необходимую операцию памяти, используя загруженное значение регистра MAR (или MDR для записи).

Нужно обсудить еще один аспект микроархитектуры Mic-4: микропереходы. Некоторым командам IJVM нужен условный переход, который осуществляется с помощью бита N. Когда происходит такой переход, конвейер не может продолжать работу. Именно поэтому нам пришлось добавить в микрооперацию бит перехода. Когда в блок формирования очереди поступает микрооперация с таким битом, блок воздерживается от передачи сигнала о получении данных блоку декодирования. В результате машина будет простаивать до тех пор, пока этот переход не разрешится.

Предположительно, некоторые команды IJVM, не зависящие от этого перехода, уже переданы в блок декодирования, но не в блок формирования очереди, поскольку он еще не выдал сигнал о получении. Чтобы разобраться в этой путанице и вернуться к нормальной работе, требуется специальное аппаратное обеспечение и особые механизмы, но мы не будем рассматривать их в этой книге. Э. Дейкстра, написавший знаменитую работу «GOTO Statement Considered Harmful» («Выражение GOTO губительно»), был прав.

Мы начали с микроархитектуры Mic-1 и, пройдя довольно долгий путь, закончили микроархитектурой Mic-4. Микроархитектура Mic-1 представляла собой очень простой вариант аппаратного обеспечения, поскольку практически все управление осуществлялось программным обеспечением. Микроархитектура Mic-4 является конвейеризированной структурой с семью стадиями и более сложным аппаратным обеспечением. Данный конвейер изображен на рис. 4.24. Цифры в кружочках соответствуют компонентам рис. 4.23. Микроархитектура Mic-4 автоматически вызывает заранее поток байтов из памяти, декодирует его в команды IJVM, превращает их в последовательность операций с помощью ПЗУ и применяет их по назначению. Первые три стадии конвейера при желании можно связать с задающим генератором тракта данных, но работа будет происходить не в каждом цикле. Например, блок выборки команд совершенно точно не может передавать новый код операции блоку декодирования в каждом цикле, поскольку выполнение команды IJVM занимает несколько циклов и очередь быстро переполнится.

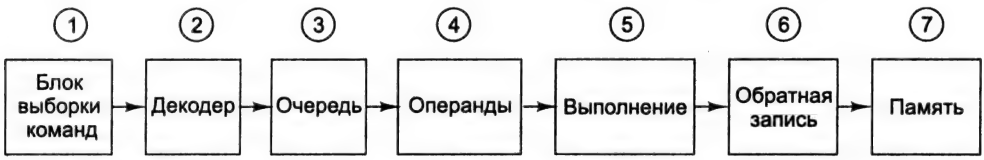


Рис. 4.24. Конвейер Mic-4

В каждом цикле значения регистров MIR перемещаются, а микрооперация, находящаяся в начале очереди, копируется в регистр MIR1. Затем сигналы управления из всех четырех регистров MIR передаются по тракту данных, вызывая определенные действия. Каждый регистр MIR контролирует отдельную часть тракта данных и, следовательно, различные микрошаги.

В данной разработке содержится конвейеризированный процессор. Благодаря этому отдельные шаги становятся очень короткими, а тактовая частота — высокой. Многие процессоры проектируются именно таким образом, особенно те, которым приходится выполнять старый набор команд (CISC). Например, процессор Pentium II в некоторых аспектах сходен с микроархитектурой Mic-4, как мы увидим позднее в этой главе.

Увеличение производительности

Все производители компьютеров хотят, чтобы их системы работали как можно быстрее. В этом разделе мы рассмотрим ряд передовых технологий для повышения производительности системы (в первую очередь процессора и памяти), которые исследуются в настоящее время. Поскольку в компьютерной промышленности наблюдается огромное количество конкурентов, между появлением новой идеи о том, как повысить скорость работы компьютера, и воплощением этой идеи обычно проходит очень небольшой период времени. Следовательно, большинство идей, которые мы сейчас будем обсуждать, уже применяются в производстве.

Усовершенствования, которые мы будем обсуждать, распадаются на две категории: усовершенствование реализации и усовершенствование архитектуры.

Усовершенствования реализации — это такие способы построения нового процессора и памяти, после применения которых система работает быстрее, но архитектура при этом не меняется. Изменение реализации без изменения архитектуры означает, что старые программы будут работать на новой машине, а это очень важно для успешной продажи. Чтобы усовершенствовать реализацию, можно, например, использовать более быстрый задающий генератор, но это не единственный способ. Отметим, что улучшение производительности от компьютера 80386 к 80486, Pentium, Pentium Pro, а затем Pentium II происходило без изменения архитектуры.

Однако некоторые типы усовершенствований можно осуществить только путем изменения архитектуры. Иногда, например, нужно добавить новые команды или регистры, причем таким образом, чтобы старые программы могли работать на новых моделях. В этом случае для достижения полной производительности программное обеспечение должно быть изменено или, по крайней мере, заново скомпилировано на новом компиляторе.

Однако один раз в несколько десятилетий разработчики понимают, что старая архитектура уже никуда не годится и что единственный способ развивать технологии дальше — начать все заново. Таким революционным скачком было появление RISC в 80-х годах, и следующий прорыв уже приближается. Мы рассмотрим наш пример (Intel IA-64) в главе 5.

Далее в этом разделе мы расскажем о четырех различных технологиях увеличения производительности процессора. Начнем мы с трех установившихся способов усовершенствования реализации, а затем перейдем к методу, для которого требуется поддержка архитектуры. Это кэш-память, прогнозирование ветвления, исполнение с изменением последовательности, подмена регистров и спекулятивное исполнение.

Кэш-память

Одним из самых важных вопросов при разработке компьютеров было и остается построение такой системы памяти, которая могла бы передавать операнды процессору с той же скоростью, с которой он их обрабатывает. Быстрый рост скорости работы процессора, к сожалению, не сопровождается столь же высоким ростом скорости работы памяти. Относительно процессора память работает все медленнее и медленнее с каждым десятилетием. С учетом огромной важности основной памяти эта ситуация сильно ограничивает развитие систем с высокой производительностью и направляет исследование таким путем, чтобы обойти проблему очень низкой по сравнению с процессором скорости работы памяти. И, откровенно говоря, эта ситуация ухудшается с каждым годом.

Современные процессоры предъявляют определенные требования к системе памяти и относительно времени ожидания (задержки в доставке операнда), и относительно пропускной способности (количества данных, передаваемых в единицу времени). К сожалению, эти два аспекта системы памяти сильно расходятся. Обычно с увеличением пропускной способности увеличивается время ожидания. Например, технологии конвейеризации, которые используются в микроархитектуре Mic-3, можно применить к системе памяти, при этом запросы памяти будут

обрабатываться более рационально, с перекрытием. Но, к сожалению, как и в микроархитектуре Mic-3, это приводит к увеличению времени ожидания отдельных операций памяти. С увеличением скорости задающего генератора становится все сложнее обеспечить такую систему памяти, которая может передавать операнды за один или два цикла.

Один из способов решения этой проблемы — добавление кэш-памяти. Как мы говорили в разделе «Кэш-память» главы 2, кэш-память содержит наиболее часто используемые слова, что повышает скорость доступа к ним. Если достаточно большой процент нужных слов находится в кэш-памяти, время ожидания может сильно сократиться.

Одной из самых эффективных технологий одновременного увеличения пропускной способности и уменьшения времени ожидания является применение нескольких блоков кэш-памяти. Основная технология — введение отдельной кэш-памяти для команд и отдельной для данных (**разделенной кэш-памяти**). Такая кэш-память имеет несколько преимуществ. Во-первых, операции могут начинаться независимо в каждой кэш-памяти, что удваивает пропускную способность системы памяти. Именно по этой причине в микроархитектуре Mic-1 нам понадобились два отдельных порта памяти: особый порт для каждой кэш-памяти. Отметим, что каждая кэш-память имеет независимый доступ к основной памяти.

В настоящее время многие системы памяти гораздо сложнее этих. Между разделенной кэш-памятью и основной памятью часто помещается **кэш-память второго уровня**. Вообще говоря, может быть три и более уровней кэш-памяти, поскольку требуются более продвинутые системы. На рис. 4.25 изображена система с тремя уровнями кэш-памяти. Прямо на микросхеме центрального процессора находится небольшая кэш-память для команд и небольшая кэш-память для данных, обычно от 16 до 64 Кбайт. Есть еще кэш-память второго уровня, которая расположена не на самой микросхеме процессора, а рядом с ним в том же блоке. Кэш-память второго уровня соединяется с процессором через высокоскоростной тракт данных. Эта кэш-память обычно не является разделенной и содержит смесь данных и команд. Ее размер — от 512 Кбайт до 1 Мбайт. Кэш-память третьего уровня находится на той же плате, что и процессор, и обычно состоит из статического ОЗУ в несколько мегабайтов, которое функционирует гораздо быстрее, чем динамическое ОЗУ основной памяти. Обычно все содержимое кэш-памяти первого уровня находится в кэш-памяти второго уровня, а все содержимое кэш-памяти второго уровня находится в кэш-памяти третьего уровня.

Существует два типа локализации адресов. Работа кэш-памяти зависит от этих типов локализации. **Пространственная локализация** основана на вероятности, что в скором времени появится потребность обратиться к ячейкам памяти, которые расположены рядом с недавно вызванными ячейками. Исходя из этого наблюдения в кэш-память переносится больше данных, чем требуется в данный момент. **Временная локализация** имеет место, когда недавно запрашиваемые ячейки запрашиваются снова. Это может происходить, например, с ячейками памяти, находящимися рядом с вершиной стека или с командами внутри цикла. Принцип временной локализации используется при выборе того, какие элементы выкинуть из

кэш-памяти в случае промаха кэш-памяти. Обычно отбрасываются те элементы, к которым давно не было обращений.



Рис. 4.25. Система с тремя уровнями кэш-памяти

Во всех типах кэш-памяти используется следующая модель. Основная память разделяется на блоки фиксированного размера, которые называются **строками кэш-памяти**. Строка кэш-памяти состоит из нескольких последовательных байтов (обычно от 4 до 64). Строки нумеруются, начиная с 0, то есть если размер строки составляет 32 байта, то строка 0 — это байты с 0 по 31, строка 1 — байты с 32 по 63 и т. д. В любой момент несколько строк находится в кэш-памяти. Когда происходит обращение к памяти, контроллер кэш-памяти проверяет, есть ли нужное слово в данный момент в кэш-памяти. Если есть, то можно сэкономить время, требуемое на доступ к основной памяти. Если данного слова в кэш-памяти нет, то какая-либо строка из нее удаляется, а вместо нее помещается нужная строка из основной памяти или из кэш-памяти более низкого уровня. Существует множество вариаций данной схемы, но в их основе всегда лежит идея держать в кэш-памяти как можно больше часто используемых строк, чтобы число успешных обращений к кэш-памяти было максимальным.

Кэш-память прямого отображения

Самый простой тип кэш-памяти — это **кэш-память прямого отображения**. Пример одноуровневой кэш-памяти прямого отображения показан на рис. 4.26, *а*. Данная кэш-память содержит 2048 элементов. Каждый элемент (ряд) может вмещать ровно одну строку из основной памяти. Если размер строки кэш-памяти 32 байта

(для этого примера), кэш-память может вмещать 64 Кбайт. Каждый элемент кэш-памяти состоит из трех частей:

1. Бит достоверности указывает, есть ли достоверные данные в элементе или нет. Когда система загружается, все элементы маркируются как недостоверные.
2. Поле «Тег» состоит из уникального 16-битного значения, указывающего соответствующую строку памяти, из которой поступили данные.
3. Поле «Данные» содержит копию данных памяти. Это поле вмещает одну строку кэш-памяти в 32 байта.

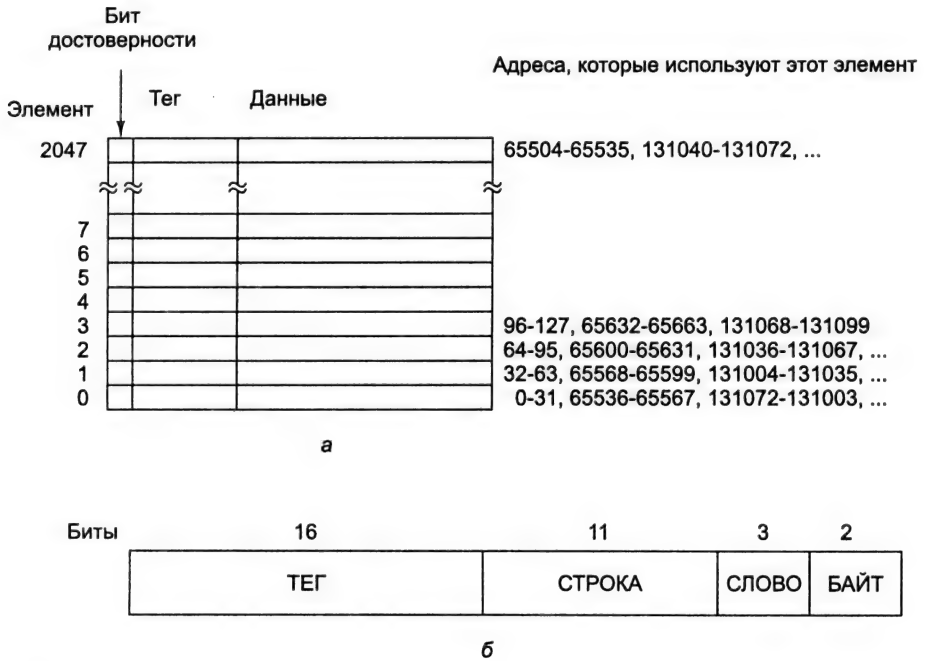


Рис. 4.26. Кэш-память прямого отображения (*а*); 32-битный виртуальный адрес (*б*)

В кэш-памяти прямого отображения данное слово может храниться только в одном месте. Если дан адрес слова, то в кэш-памяти его можно искать только в одном месте. Если его нет на этом определенном месте, значит, его вообще нет в кэш-памяти. Для хранения и удаления данных из кэш-памяти адрес разбивается на 4 компонента, как показано на рис. 4.26, *б*:

1. Поле «ТЕГ» соответствует битам, сохраненным в поле «Тег» элемента кэш-памяти.
2. Поле «СТРОКА» указывает, какой элемент кэш-памяти содержит соответствующие данные, если они есть в кэш-памяти.
3. Поле «СЛОВО» указывает, на какое слово в строке производится ссылка.
4. Поле «БАЙТ» обычно не используется, но если требуется только один байт, поле сообщает, какой именно байт в слове нужен. Для кэш-памяти, поддерживающей только 32-битные слова, это поле всегда будет содержать 0.

Когда центральный процессор выдает адрес памяти, аппаратное обеспечение выделяет из этого адреса 11 битов поля «СТРОКА» и использует их для поиска в кэш-памяти одного из 2048 элементов. Если этот элемент действителен, то производится сравнение поля «Тег» основной памяти и поля «Тег» кэш-памяти. Если поля равны, это значит, что в кэш-памяти есть слово, которое запрашивается. Такая ситуация называется **удачным обращением в кэш-память**. В случае удачного обращения слово берется прямо из кэш-памяти, и тогда не нужно обращаться к основной памяти. Из элемента кэш-памяти берется только нужное слово. Остальная часть элемента не используется. Если элемент кэш-памяти недействителен (недостоверен) или поля «Тег» не совпадают, то нужного слова нет в памяти. Такая ситуация называется **промахом кэш-памяти**. В этом случае 32-байтная строка вызывается из основной памяти и сохраняется в кэш-памяти, заменяя тот элемент, который там был. Однако если существующий элемент кэш-памяти изменяется, его нужно записать обратно в основную память до того, как он будет отброшен.

Несмотря на сложность решения, доступ к нужному слову может быть чрезвычайно быстрым. Поскольку известен адрес, известно и точное нахождение слова, *если оно имеется в кэш-памяти*. Это значит, что можно считывать слово из кэш-памяти и доставлять его процессору и одновременно с этим устанавливать, правильное ли это слово (путем сравнения полей «Тег»). Поэтому процессор в действительности получает слово из кэш-памяти одновременно или даже до того, как станет известно, требуемое это слово или нет.

При такой схеме последовательные строки основной памяти помещаются в последовательные элементы кэш-памяти. Фактически в кэш-памяти может храниться до 64 Кбайт смежных данных. Однако две строки, адреса которых различаются ровно на 64 К (65, 536 байт) или на любое целое кратное этому числу, не могут одновременно храниться в кэш-памяти (поскольку они имеют одно и то же значение в поле «СТРОКА»). Например, если программа обращается к данным с адресом X, а затем выполняет команду, которой требуются данные с адресом X+ 65, 536 (или с любым другим адресом в той же строке), вторая команда требует перезагрузки элемента кэш-памяти. Если это происходит достаточно часто, то могут возникнуть проблемы. В действительности, если кэш-память плохо работает, то лучше бы вообще не было кэш-памяти, поскольку при каждой операции с памятью считывается целая строка, а не одно слово.

Кэш-память прямого отображения — это самый распространенный тип кэш-памяти, и она достаточно эффективна, поскольку коллизии, подобные описанной выше, случаются крайне редко[1] или вообще не случаются. Например, очень хороший компилятор может учитывать подобные коллизии при размещении команд и данных в памяти. Отметим, что указанный выше случай не произойдет в системе, где команды и данные находятся раздельно, поскольку конфликтующие запросы будут обслуживаться разными блоками кэш-памяти. Таким образом, мы видим второе преимущество наличия двух блоков кэш-памяти вместо одного: большая гибкость при разрешении конфликтных ситуаций.

[1] На самом деле подобные коллизии не столь уж и редки из-за того, что при страничном способе организации виртуальной памяти и организации параллельного выполнения нескольких задач страницы как бы «перемешиваются». Разбиение программы на страницы осуществляется случайным образом, поэтому и «локальность кода» может быть нарушена. — *Примеч. научн. ред.*

Ассоциативная кэш-память с множественным доступом

Как было сказано выше, различные строки основной памяти конкурируют за право занять одну и ту же область в кэш-памяти. Если программе, которая применяет кэш-память, изображенную на рис. 4.26, *а*, часто требуются слова с адресами 0 и 65 536, то будут иметь место постоянные конфликты и каждое обращение потенциально повлечет за собой вытеснение какой-то определенной строки кэш-памяти. Чтобы разрешить эту проблему, нужно сделать так, чтобы в каждом элементе кэш-памяти помещалось по две и более строк. Кэш-память с n возможными элементами для каждого адреса называется **n-входовой ассоциативной кэш-памятью**. Четырехвходовая ассоциативная кэш-память изображена на рис. 4.27.



Рис. 4.27. Четырехвходовая ассоциативная кэш-память

Ассоциативная кэш-память с множественным доступом по сути гораздо сложнее, чем кэш-память прямого отображения, поскольку хотя элемент кэш-памяти и можно вычислить из адреса основной памяти, требуется проверить n элементов кэш-памяти, чтобы узнать, есть ли там нужная нам строка. Тем не менее практика показывает, что двувходовая или четырехвходовая ассоциативная кэш-память дает хороший результат, поэтому внедрение этих дополнительных схем вполне оправданно.

Использование ассоциативной кэш-памяти с множественным доступом ставит разработчика перед выбором. Если нужно поместить новый элемент в кэш-память, какой именно из старых элементов нужно убрать? Для многих целей хорошо подходит алгоритм **LRU (Least Recenly Used — алгоритм удаления наиболее давно использовавшихся элементов)**. Имеется определенный порядок каждого набора ячеек, которые могут быть доступны из данной ячейки памяти. Всякий раз, когда осуществляется доступ к любой строке, в соответствии с алгоритмом список обновляется и маркируется элемент, к которому произведено последнее обращение. Когда требуется заменить какой-нибудь элемент, убирается тот, который находится в конце списка, то есть тот, который использовался давно по сравнению со всеми другими.

Возможна также 2048-входовая ассоциативная кэш-память, которая содержит один набор из 2048 элементов. Здесь все адреса памяти располагаются в этом наборе, поэтому при поиске требуется сравнивать нужный адрес со всеми 2048 тегами в кэш-памяти. Отметим, что для этого каждый элемент кэш-памяти должен содержать специальную логическую схему. Поскольку поле «СТРОКА» в данный момент имеет длину 0, поле «ТЕГ» — это весь адрес за исключением полей «СЛОВО» и «БАЙТ». Более того, когда строка кэш-памяти замещается, все 2048 ячеек являются возможными кандидатами на смену. Для сохранения упорядоченного списка потребовался бы громоздкий учет использования системных ресурсов, поэтому применение алгоритма LRU становится недопустимым. (Помните, что этот список следует обновлять при каждой операции с памятью.) Интересно, что кэш-память с высокой ассоциативностью часто не сильно превосходит по производительности кэш-память с низкой ассоциативностью, а в некоторых случаях работает даже хуже. Поэтому ассоциативность выше четырех встречается редко.

Наконец, особой проблемой для кэш-памяти является запись. Когда процессор записывает слово, а это слово находится в кэш-памяти, он, очевидно, должен или обновить слово, или отбросить данный элемент кэш-памяти. Практически во всех разработках используется обновление кэш-памяти. А что же можно сказать об обновлении копии в основной памяти? Эту операцию можно отложить на потом до того момента, когда строка кэш-памяти будет готова к замене алгоритмом LRU. Выбор труден, и ни одно из решений не является предпочтительным. Немедленное обновление элемента основной памяти называется **сквозной записью**. Этот подход обычно гораздо проще реализуется, и к тому же, он более надежен, поскольку современная память всегда может восстановить предыдущее состояние, если произошла ошибка. К сожалению, при этом требуется передавать больший поток информации к памяти, поэтому в более сложных проектах стремятся использовать альтернативный подход — **обратную запись**.

С процессом записи связана еще одна проблема: а что происходит, если нужно записать что-либо в ячейку, которая в текущий момент не находится в кэш-памяти? Должны ли данные переноситься в кэш-память или просто записываться в основную память? И снова ни один из ответов не является во всех отношениях лучшим. В большинстве разработок, в которых применяется обратная запись, данные переносятся в кэш-память. Эта технология называется **заполнением по записи** (write allocation). С другой стороны, в тех разработках, где применяется сквозная запись, обычно элемент в кэш-память при записи не помещается, поскольку эта возможность усложняет разработку. Заполнение по записи полезно только в том случае, если имеют место повторные записи в одно и то же слово или в разные слова в пределах одной строки кэш-памяти.

Прогнозирование ветвления

Современные компьютеры сильно конвейеризированы. Конвейер, изображенный на рис. 4.25, имеет семь стадий; более сложно организованные компьютеры содержат конвейеры с десятью и более стадиями. Конвейеризация лучше работает с линейным кодом, поэтому блок выборки команд может просто считывать последовательные слова из памяти и отправлять их в блок декодирования заранее, еще до того, как они понадобятся.

Единственная проблема состоит в том, что эта модель совершенно не реалистична. Программы вовсе не являются последовательностями линейного кода. В них полно команд перехода. Рассмотрим простые утверждения листинга 4.4. Переменная i сравнивается с 0 (вероятно, это самый распространенный тест на практике). В зависимости от результата другой переменной, k, присваивается одно из двух возможных значений.

Листинг 4.4. Фрагмент программы

```
if(i==0)
        k=1;
else
        k=2;
```

Возможный перевод на язык ассемблера показан в листинге 4.5. Язык ассемблера мы будем рассматривать позже в этой книге, и детали сейчас не важны, но при определенных машине и компиляторе программа, более или менее похожая на программу листинга 4.5, вполне возможна. Первая команда сравнивает переменную i с 0. Вторая совершает переход к `Else`, если i не равно 0. Третья команда присваивает значение 1 переменной k. Четвертая команда совершает переход к следующему высказыванию программы. Компилятор поместил там метку `Next`. Пятая команда присваивает значение 2 переменной k.

Листинг 4.5. Перевод программы листинга 4.4 на язык ассемблер

```
        CMP i, 0    ; сравнение i с 0
        BNE Else    ; переход к Else, если они не равны
Then:   MOV k, 1    ; присваивание значения 1 переменной k
        BR Next     ; безусловный переход к Next
Else:   MOV k, 2    ; присваивание значения 2 переменной k
Next:
```

Мы видим, что две из пяти команд являются переходами. Более того, одна из них, `BNE`, — это условный переход (переход, который осуществляется тогда и только тогда, когда выполняется определенное условие, в данном случае это равенство двух операндов предыдущей команды `CMP`). Самый длинный линейный код состоит здесь из двух команд. Вследствие этого вызывать команды с высокой скоростью для передачи в конвейер очень трудно.

На первый взгляд может показаться, что безусловные переходы, например команда `BR Next` в листинге 4.5, не влекут за собой никаких проблем. Вообще говоря, в данном случае нет никакой двусмысленности в том, куда дальше идти. Почему же блок выборки команд не может просто продолжать считывать команды из целевого адреса (то есть из того места, куда будет затем осуществлен переход)?

Сложность объясняется самой природой конвейеризации. На рис. 4.23, например, мы видим, что декодирование происходит на второй стадии. Следовательно, блоку выборки команд приходится решать, откуда вызывать следующую команду еще до того, как он узнает, команду какого типа он только что вызвал. Только в следующем цикле он сможет узнать, что получил команду безусловного перехода, и до этого момента он уже начал вызывать команду, следующую за безусловным переходом. Вследствие этого существенная часть конвейеризированных машин (например, UltraSPARC II) обладает таким свойством, что сначала выполняется команда, *следующая после* безусловного перехода, хотя по логике вещей этого не

должно быть. Позиция после перехода называется **отсрочкой ветвления**. Pentium II (а также машина, используемая в листинге 4.5) не обладает таким качеством, но обойти эту проблему путем усложнения внутреннего устройства чрезвычайно тяжело. Оптимизирующий компилятор постарается найти какую-нибудь полезную команду, чтобы поместить ее в отсрочку ветвления, но часто ничего подходящего нет, поэтому компилятор вынужден вставлять туда команду `NOP`. Это сохраняет правильность программы, но зато программа становится больше по объему и работает медленнее.

С условными переходами дело обстоит еще хуже. Во-первых, они тоже содержат отсрочки ветвления, а во-вторых, блок выборки команд узнает, откуда нужно считывать команду, гораздо позже. Первые конвейеризированные машины просто **простаивали** до тех пор, пока не становилось известно, нужно ли совершать переход или нет. Простаивание по три или четыре цикла при каждом условном переходе, особенно если 20% команд являются условными переходами, сильно снижает производительность.

Поэтому большинство машин прогнозируют, будет производиться условный переход, который встретился на пути, или нет. Для этого, например, можно предполагать, что все условные переходы назад будут осуществляться, а все условные переходы вперед не будут. Что касается первой части предположения, причина такого выбора состоит в том, что переходы назад часто помещаются в конце цикла. Большинство циклов выполняется много раз, поэтому переход к началу цикла будет встречаться очень часто.

Со второй частью данного предположения дело обстоит сложнее. Некоторые переходы вперед осуществляются в случае обнаружения ошибки в программном обеспечении (например, файл не может быть открыт). Ошибки случаются редко, поэтому в большинстве случаев подобные переходы не происходят. Естественно, существует множество переходов вперед, не связанных с ошибками, поэтому процент успеха здесь не так высок, как в переходах назад. Однако это правило по крайней мере лучше, чем ничего.

Если переход правильно предсказан, то ничего особенного делать не нужно. Просто продолжается выполнение программы. Проблема возникает в том случае, когда переход предсказан неправильно. Вычислить, куда нужно перейти, и перейти именно туда несложно. Самое сложное — отменить команды, которые уже выполнены, но которые не нужно было выполнять.

Существует два способа отмены команд. Первый способ — продолжать выполнять команды, вызванные после спрогнозированного условного перехода, до тех пор, пока одна из этих команд не попытается изменить состояние машины (например, сохранить значение в регистре). Тогда вместо того, чтобы перезаписывать этот регистр, нужно поместить вычисленное значение во временный (скрытый) регистр, а затем, когда уже станет известно, что прогноз был правильным, просто скопировать это значение в обычный регистр. Второй способ — записать значение любого регистра, который, вероятно, скоро будет переписан (например, в скрытый временный регистр), поэтому машина может вернуться в предыдущее состояние в случае неправильно предсказанного перехода. Реализация обоих способов очень сложна и требует громоздкого учета использования системных ресурсов. А если встречается второй условный переход до того, как стало известно, был ли первый условный переход предсказан правильно, все может совершенно запутаться.

Динамическое прогнозирование ветвления

Ясно, что точные прогнозы очень ценны, поскольку это позволяет процессору работать с полной скоростью. В настоящее время проводится множество исследований, целью которых является усовершенствование алгоритмов прогнозирования ветвления (например, [32, 70, 108, 125, 138, 163]). Один из подходов — хранить специальную таблицу (в особом аппаратном обеспечении), в которую центральный процессор записывает условные переходы, когда они встречаются, и там их можно искать, когда они снова появятся. Простейшая версия такой схемы показана на рис. 4.28, *а*. В данном случае эта таблица содержит одну ячейку для каждой команды условного перехода. В ячейке находится адрес команды перехода, а также бит, который указывает, был ли сделан переход, когда эта команда встретилась последний раз. Прогноз состоит в том, что программа пойдет тем же путем, каким она пошла в прошлый раз после этой команды перехода. Если прогноз неверен, бит в таблице меняется.

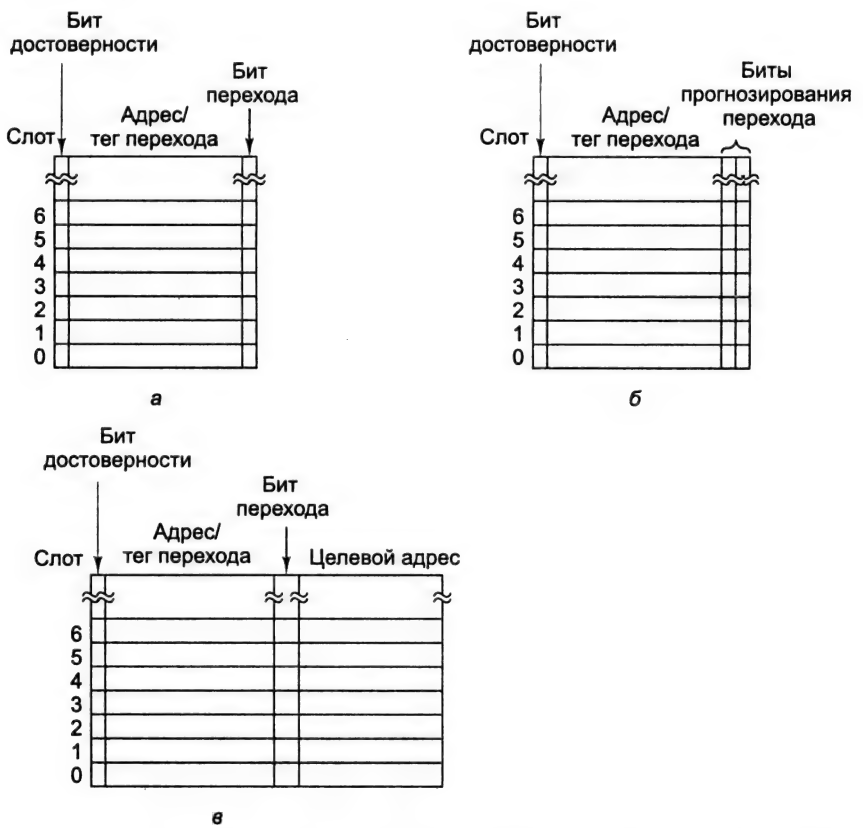


Рис. 4.28. Таблица динамики ветвлений с 1-битным указателем перехода (*а*); таблица динамики ветвлений с 2-битным указателем перехода (*б*); соответствие между адресом команды перехода и целевым адресом (*в*)

Существует несколько способов организации данной таблицы. В действительности точно такие же способы используются при организации кэш-памяти.

Рассмотрим машину с 32-битными командами, которые расположены таким образом, что два младших бита каждого адреса памяти — 00. Таблица содержит 2^n ячеек (строк). Из команды перехода можно извлечь n+2 младших бита и осуществить сдвиг вправо на два бита. Это n-битное число можно использовать в качестве индекса в таблице, где проверяется, совпадает ли адрес, сохраненный там, с адресом перехода. Как и в случае с кэш-памятью, здесь нет необходимости сохранять n+2 младших бита, поэтому их можно опустить (то есть сохраняются только старшие адресные биты — тег). Если адреса совпали, бит прогнозирования используется для предсказания перехода. Если тег неправильный или элемент недействителен, значит, имеет место несовпадение. В этом случае можно применять правило перехода вперед/назад.

Если таблица динамики переходов содержит, скажем, 4096 элементов, то адреса 0, 16384, 32768, ... будут конфликтовать; аналогичная проблема встречается и при работе с кэш-памятью. Здесь возможно такое же решение: двухальтернативный, четырехальтернативный, n-альтернативный ассоциативный элемент. Как и у кэш-памяти, предельный случай — один n-альтернативный ассоциативный элемент.

При достаточно большом размере таблицы и достаточной ассоциативности эта схема хорошо работает в большинстве ситуаций. Тем не менее систематически встречается одна проблема. Когда происходит выход из цикла, переход в конце будет предсказан неправильно, и, что еще хуже, этот неправильный прогноз изменит бит в таблице, который теперь будет указывать, что переход совершать не надо. В следующий раз, когда опять будет выполняться цикл, переход в конце первого прохождения цикла будет спрогнозирован неправильно. Если цикл находится внутри другого цикла или внутри часто вызываемой процедуры, эта ошибка будет повторяться слишком часто.

Для устранения такой ситуации мы немного изменим метод, чтобы прогноз менялся только после двух последовательных неправильных предсказаний. Такой подход требует наличия двух предсказывающих битов в таблице: один указывает, предполагается ли совершить переход или нет, а второй указывает, что было сделано в прошлый раз. Таблица показана на рис. 4.28, *б*.

Этот алгоритм можно представить в виде конечного автомата с четырьмя состояниями (рис. 4.29). После ряда последовательных успешных предсказаний «перехода нет» конечный автомат будет находиться в состоянии 00 и в следующий раз также прогнозировать, что «перехода нет». Если этот прогноз неправильный, автомат переходит в состояние 01, но в следующий раз все равно предсказывает отсутствие перехода. Только в том случае, если это последнее предсказание ошибочно, конечный автомат перейдет в состояние 11 и будет все время прогнозировать наличие перехода. Фактически, левый бит — это прогноз, а правый бит — это то, что было сделано в прошлый раз (то есть был ли совершен переход). В данной разработке используется только 2 специальных бита, но возможно применение и 4, и 8 битов.

Это не первый конечный автомат, который мы рассматриваем. На рис. 4.19 тоже изображен конечный автомат. На самом деле все наши микропрограммы можно считать конечными автоматами, поскольку каждая строка представляет особое состояние, в котором может находиться автомат, с четко определенными переходами к конечному набору других состояний. Конечные автоматы очень широко используются во всех аспектах разработки аппаратного обеспечения.



Рис. 4.29. Двубитный конечный автомат для прогнозирования переходов

До сих пор мы предполагали, что цель каждого условного перехода известна. Обычно или в явном виде давался адрес, к которому нужно перейти (он содержался прямо в самой команде), или было известно смещение относительно текущей команды (то есть число со знаком, которое нужно было прибавить к счетчику команд). Часто это предположение имеет силу, но некоторые команды условного перехода вычисляют целевой адрес, выполняя определенные арифметические действия над значениями регистров, а затем уже переходят туда. Даже если взять конечный автомат, изображенный на рис. 4.29, который точно прогнозирует переходы, такой прогноз будет не нужен, поскольку целевой адрес неизвестен. Один из возможных выходов из подобной ситуации — сохранить в таблице адрес, к которому был осуществлен переход в прошлый раз, как показано на рис. 4.28, *в*. Тогда, если в таблице указано, что в прошлый раз, когда встретилась команда перехода по адресу 516, переход был совершен в адрес 4000, и если сейчас предсказывается совершение перехода, то целевым адресом снова будет 4000.

Еще один подход к прогнозированию ветвления — следить, были ли совершены последние k условных переходов, независимо от того, какие это были команды [108]. Это k-битное число, которое хранится в **сдвиговом регистре динамики переходов**, затем сравнивается параллельно со всеми элементами таблицы с k-битным ключом, и в случае совпадения применяется то предсказание, которое найдено в этом элементе. Удивительно, но эта технология работает достаточно хорошо.

Статическое прогнозирование ветвления

Все технологии прогнозирования ветвления, которые обсуждались до сих пор, являются динамическими, то есть выполняются во время работы программы. Они также приспосабливаются к текущему поведению программы, и это их положительное качество. Отрицательной стороной этих технологий является то, что они требуют специализированного и дорогостоящего аппаратного обеспечения, а также наличия очень сложных микросхем.

Можно пойти другим путем и призвать на помощь компилятор. Когда компилятор получает такое выражение, как

```
for (i=0; i < 1000000; i++) { … }
```

это знает, что переход в конце цикла будет происходить практически всегда. Если бы только был способ сообщить это аппаратному обеспечению, можно было бы избавиться от огромного количества работы.

Хотя это связано с изменением архитектуры (а не только с вопросом реализации), в некоторых машинах, например UltraSPARC II, имеется еще один набор команд условного перехода помимо обычных (которые нужны для обратной совместимости). Новые команды содержат бит, по которому компилятор определяет, совершать переход или не совершать. Когда встречается такой бит, блок выборки команд просто делает то, что ему сказано. Более того, нет необходимости тратить драгоценное пространство в таблице предыстории переходов для этих команд, что сокращает количество конфликтных ситуаций.

Наконец, наша последняя технология прогнозирования ветвления основана на профилировании [37]. Это тоже статическая технология, только в данном случае программа не заставляет компилятор вычислять, какие переходы нужно совершать, а какие нет. В данном случае программа действительно выполняется, а ветвления фиксируются. Эта информация поступает в компилятор, который затем использует специальные команды условного перехода для того, чтобы сообщить аппаратному обеспечению, что нужно делать.

Исполнение с изменением последовательности и подмена регистров

Большинство современных процессоров являются и конвейеризированными и суперскалярными, как показано на рис. 2.5. Это значит, что там есть блок выборки команд, который заранее вызывает команды из памяти и передает их в блок декодирования. Блок декодирования, в свою очередь, передает декодированные команды в соответствующие функциональные блоки для выполнения. В некоторых случаях этот блок может разбивать отдельные команды на микрооперации, перед тем как отправить их в функциональные блоки.

Ясно, что самым простым является компьютер, в котором все команды выполняются в том порядке, в котором они вызываются из памяти (предполагается, что прогнозирование переходов всегда оказывается верным). Однако такое последовательное выполнение не всегда дает оптимальную производительность из-за взаимной зависимости команд. Если команде требуется значение, которое вычисляется предыдущей командой, вторая команда не может начать выполняться, пока первая не выдаст нужную величину. В такой ситуации реальной взаимозависимости второй команде приходится ждать. Существуют и другие виды взаимозависимостей, но о них мы поговорим позже.

Чтобы обойти эти проблемы и достичь лучшей производительности, некоторые процессоры пропускают взаимозависимые команды и переходят к следующим (независимым) командам. Думаю, не нужно говорить, что алгоритм распределения команд должен давать такой же результат, как если бы все команды выполнялись в том порядке, в котором они написаны. А теперь продемонстрируем на конкретном примере, как происходит переупорядочение команд.

Чтобы изложить основную суть проблемы, начнем с машины, которая запускает команды в том порядке, в котором они расположены в программе, и требует, чтобы выполнение команд завершалось также в порядке, соответствующем программному. Важность второго требования прояснится позднее.

Наша машина содержит 8 регистров, видимых для программиста, от R0 до R7. Все арифметические команды используют три регистра: два — для операндов и один — для результата, как и в микроархитектуре Mic-4. Мы предполагаем, что если команда декодируется в цикле n, выполнение начинается в цикле n+1. В случае с простой командой, например командой сложения или вычитания, запись обратно в выходной регистр происходит в конце цикла n+2. В случае с более сложной командой, например командой умножения, запись в регистр происходит в конце цикла n+3. Чтобы сделать наш пример реалистичным, мы позволим блоку декодирования выпускать до двух команд за цикл. Некоторые суперскалярные процессоры могут выпускать 4 или даже 6 команд за цикл.

Последовательность выполнения команд показана в табл. 4.12. В первом столбце приводится номер цикла, во втором — номер команды, а в третьем — сама команда. В четвертом столбце указано, выдача каких команд произошла (максимум две команды за цикл). Цифры в пятом столбце сообщают, какие команды завершены. Помните, что в нашем примере мы требуем, чтобы команды и запускались, и завершались в строгом порядке, поэтому выдача команды k+1 может произойти только после выдачи команды k, а результат команды k+1 не может быть записан в выходной регистр до того, как завершится выполнение команды k. Оставшиеся 16 столбцов мы обсудим ниже.

Таблица 4.12. Суперскалярный процессор с последовательной выдачей и последовательным завершением команд

Цикл	#	Команда	Выдача	Завершение	Считываемые регистры								Записываемые регистры							
					0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1	1	R3=R0*R1	1		1	1										1				
	2	R4=R0+R2	2		2	1	1									1	1			
2	3	R5=R0+R1	3		3	2	1									1	1	1		
	4	R6=R1+R4	–		3	2	1									1	1	1		
3					3	2	1									1	1	1		
4				1	2	1	1										1	1		
				2	1	1												1		
				3																
5			4			1			1										1	
	5	R7=R1*R2	5			2	1		1										1	1
6	6	R1=R0-R2	–			2	1		1										1	1
7				4		1	1													1
8				5																
9			6		1		1							1						
	7	R3=R3*R1	–		1		1							1						
10					1		1							1						
11				6																
12			7			1		1								1				
	8	R1=R4+R4	–			1		1								1				
13						1		1								1				
14						1		1								1				
15				7																
16			8						2					1						
17									2					1						
18				8																

После декодирования команды блок декодирования должен определить, запускать ли команду сразу или нет. Для этого блок декодирования должен знать состояния всех регистров. Если, например, текущей команде требуется регистр, значение которого еще не подсчитано, текущая команда не может быть выпущена, и центральный процессор должен простаивать.

Следить за состоянием регистров будет специальное устройство — **счетчик обращений (scoreboard)**, который впервые появился в CDC 6600. Счетчик обращений содержит небольшой счетчик для каждого регистра, который показывает, сколько раз этот регистр используется командами, выполняющимися в данный момент, в качестве источника. Если одновременно может выполняться максимум 15 команд, тогда будет достаточно 4-битного счетчика. Когда запускается команда, элементы счетчика обращений, соответствующие регистрам операндов, увеличиваются на 1. Когда выполнение команды завершено, соответствующие элементы счетчика уменьшаются на 1.

Счетчик обращений также содержит счетчики для регистров, которые используются в качестве пунктов назначения. Поскольку допускается только одна запись за раз, эти счетчики могут быть размером в один бит. Правые 16 столбцов табл. 4.12 демонстрируют показания счетчика обращений.

В реальных машинах счетчик обращений также следит за использованием функционального блока, чтобы избежать выдачи команды, для которой нет доступного функционального блока. Для простоты мы предполагаем, что подходящий функциональный блок всегда имеется в наличии, поэтому функциональные блоки в таблице не показаны.

В первой строке табл. 4.12 показана команда 1, которая перемножает значения регистров R0 и R1, и помещает результат в регистр R3. Поскольку ни один из этих регистров еще не используется, команда запускается, а счетчик обращений показывает, что регистры R0 и R1 считываются, а регистр R3 записывается. Ни одна из последующих команд не может записывать результат в эти регистры и не может считывать регистр R3 до тех пор, пока не завершится выполнение команды 1. Поскольку это команда умножения, она закончится в конце цикла 4. Значения счетчика обращений, приведенные в каждой строке, отражают состояние регистров после запуска команды, записанной в этой же строке. Пустые клетки соответствуют значению 0.

Поскольку рассматриваемый пример — это суперскалярная машина, которая может запускать две команды за цикл, вторая команда выдается также во время цикла 1. Она складывает значения регистров R0 и R2, а результат сохраняет в регистре R4. Чтобы определить, можно ли запускать эту команду, применяются следующие правила:

1. Если какой-нибудь операнд записывается, запускать команду нельзя (RAW-взаимозависимость).
2. Если считывается регистр результатов, запускать команду нельзя (WAR-взаимозависимость).
3. Если записывается регистр результатов, запускать команду нельзя (WAW-взаимозависимость).

Мы уже рассматривали RAW-взаимозависимости, имеющие место, когда команде в качестве источника нужно использовать результат предыдущей команды,

которая еще не завершилась. Два других типа взаимозависимостей менее серьезные. По существу, они связаны с конфликтами ресурсов. В **WAR-взаимозависимости** (Write After Read — запись после чтения) одна команда пытается перезаписать регистр, который предыдущая команда еще не закончила считывать. **WAW-взаимозависимость (Write After Write — запись после записи)** сходна с WAR-взаимозависимостью. Этого можно избежать, если вторая команда будет помещать результат где-либо в другом месте еще (возможно, временно). Если ни одна из трех упомянутых ситуаций не возникает и нужный функциональный блок доступен, то команду можно выпустить. В этом случае команда 2 использует регистр R0, который в данный момент считывается незаконченной командой, но подобное перекрытие допустимо, поэтому команда 2 может запускаться. Сходным образом команда 3 запускается во время цикла 2.

А теперь перейдем к команде 4, которая должна использовать регистр R4. К сожалению, из таблицы мы видим, что в регистр R4 в данный момент производится запись (см. строку 3 в таблице). Здесь имеет место RAW-взаимозависимость, поэтому блок декодирования простаивает до тех пор, пока регистр R4 не станет доступен. Во время простаивания блок декодирования прекращает получать команды из блока выборки команд. Когда внутренние буферы блока выборки команд заполнятся, он прекращает вызывать команды из памяти.

Следует упомянуть, что следующая команда, команда 5, не конфликтует ни с одной из завершенных команд. Ее можно было бы декодировать и выпустить, если бы в нашей разработке не требовалось, чтобы команды выдавались по порядку.

Посмотрим, что происходит в цикле 3. Команда два, а это команда сложения (два цикла), завершается в конце цикла 3. Но ее результат не может быть сохранен в регистре R4 (который тогда освободится для команды 4). Почему? Потому что данная разработка требует записи результатов в регистры в соответствии с порядком программы. Но зачем? Что плохого произойдет, если сохранить результат в регистре R4 сейчас и сделать это значение доступным?

Ответ на этот вопрос очень важен. Предположим, что команды могут завершаться в произвольном порядке. Тогда в случае прерывания будет очень сложно сохранить состояние машины так, чтобы его можно было потом восстановить. В частности, нельзя будет сказать, что все команды до какого-то адреса были выполнены, а все команды после этого адреса не были выполнены. Это называется **точным прерыванием** и является желательной характеристикой центрального процессора [99]. Сохранение результатов в произвольном порядке делает прерывания неточными, и именно поэтому в некоторых машинах требуется соблюдение жесткого порядка в завершении команд.

Вернемся к нашему примеру. В конце четвертого цикла результаты всех трех команд могут быть сохранены, поэтому в цикле 5 может быть выпущена команда 4, а также недавно декодированная команда 5. Всякий раз, когда завершается какая-нибудь команда, блок декодирования должен проверять, нет ли простаивающей команды, которую теперь уже можно выпустить.

В цикле 6 команда 6 простаивает, потому что ей нужно записать результат в регистр R1, а регистр R1 занят. Выполнение команды начинается только в цикле 9. Чтобы завершить всю последовательность из 8 команд, требуется 15 циклов из-за многочисленных ситуаций взаимозависимости, хотя аппаратное обеспечение

способно выдавать по две команды за цикл. В колонках «Выдача» и «Завершение» табл. 4.12 видно, что все команды выдаются из блока декодирования по порядку и завершаются эти команды тоже по порядку.

Рассмотрим альтернативный подход: исполнение с изменением последовательности. В такой разработке выполнение команд может начинаться в произвольном порядке и завершаться также в произвольном порядке. В табл. 4.13 показана та же последовательность из восьми команд, только теперь разрешен другой порядок выдачи команд и сохранения результатов в регистрах.

Первое различие встречается в цикле 3. Несмотря на то, что команда 4 простаивает, мы можем декодировать и запустить команду 5, поскольку она не создает конфликтной ситуации ни с одной из выполняющихся программ. Однако пропуск команд порождает новую проблему. Предположим, что команда 5 использует операнд, который вычисляется пропущенной командой 4. При текущем состоянии счетчика обращений мы этого не заметим. В результате нам придется расширить счетчик обращений, чтобы следить за записями, которые совершают пропущенные команды. Это можно сделать, добавив еще одно битовое отображение, по одному биту на регистр, для контроля за записями, которые делают простаивающие команды (эти счетчики в таблице не показаны). Правило для запуска команд следует расширить, с тем чтобы предотвратить запуск команды, операнд которой должен быть записан той предшествующей командой, что шла до нее, но была пропущена.

Таблица 4.13. Работа суперскалярного процессора с изменением последовательности запуска и завершения команд

Цикл	#	Команда	Выдача	Завершение	Считываемые регистры								Записываемые регистры							
					0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1	1	R3=R0*R1	1		1	1										1				
	2	R4=R0+R2	2		2	1	1									1	1			
2	3	R5=R0+R1	3		3	2	1									1	1	1		
	4	R6=R1+R4	–		3	2	1									1	1	1		
3	5	R7=R1*R2	5		3	3	2									1	1	1		1
	6	S1=R0–R2	6		4	3	3									1	1	1		1
				2	3	3	2									1		1		1
4			4		3	4	2		1							1		1	1	1
	7	R3=R3*S1	–		3	4	2		1							1		1	1	1
	8	S2=R4+R4	8		3	4	2		3							1		1	1	1
				1	2	3	2		3									1	1	1
				3	1	2	2		3										1	1
5				6		2	1		3					1					1	1
6			7			2	1	1	3					1		1			1	1
				4		1	1	1	2					1		1				1
				5				1	2					1		1				
				8				1								1				
7								1								1				
8								1								1				
9				7																

Теперь посмотрим на команды 6, 7 и 8 в табл. 4.12. Здесь мы видим, что команда 6 помещает вычисленное значение в регистр R1, и это значение использует-

ся командой 7. Мы также видим, что это значение больше не используется, потому что команда 8 переписывает значение регистра R1. Нет никакой надобности использовать регистр R1 для хранения результата команды 6. Еще хуже то, что далеко не лучшим является выбор R1 в качестве промежуточного регистра, хотя с точки зрения программиста, привыкшего к идее последовательного выполнения команд без перекрытий, этот выбор является самым разумным.

В таблице 4.12 мы ввели новый метод для решения этой проблемы: **подмена регистров**. Блок декодирования меняет регистр R1 в команде 6 (цикл 3) и в команде 7 (цикл 4) на скрытый регистр S1, который невидим для программиста. Теперь команда 6 может запускаться одновременно с командой 5. Современные процессоры содержат десятки скрытых регистров, которые используются для процедуры подмены. Такая технология часто устраняет WAR- и WAW-взаимозависимости.

В команде 8 мы снова применяем подмену регистров. На этот раз регистр R1 переименовывается в S2, поэтому операция сложения может начаться до того, как регистр R1 освободится, а освободится он только в конце цикла 6. Если окажется, что результат в этот момент должен быть в регистре R1, содержимое регистра S2 всегда можно скопировать туда. Еще лучше то, что все будущие команды, которым нужен этот результат, могут в качестве источника использовать регистры, переименованные в тот регистр, где действительно хранится нужное значение. В любом случае выполнение команды 8 начнется раньше.

В настоящих (не гипотетических) компьютерах подмена регистров происходит с многократным вложением. Существует множество скрытых регистров и таблица, в которой показывается соответствие видимых для программиста регистров и скрытых регистров. Например, чтобы найти местоположение регистра R0, нужно обратиться к элементу 0 этой таблицы. На самом деле реального регистра R0 нет, а есть только связь между именем R0 и одним из скрытых регистров. Эта связь часто меняется во время выполнения программы, чтобы избежать взаимозависимостей.

Обратите внимание на четвертый и пятый столбцы табл. 4.12. Вы видите, что команды запускаются не по порядку и завершаются также не по порядку. Вывод весьма прост: изменяя последовательность выполнения команд и подменяя регистры, мы можем ускорить процесс вычисления почти в два раза.

Спекулятивное выполнение

В предыдущем разделе мы ввели понятие переупорядочения команд. Эта процедура нужна для улучшения производительности. В действительности имелось в виду переупорядочение команд в пределах одного базового элемента программы. Рассмотрим этот аспект подробнее.

Компьютерные программы можно разбить на **базовые элементы**, каждый из которых представляет собой линейную последовательность команд с точкой входа в начале и точкой выхода в конце. Базовый элемент не содержит никаких управляющих структур (например, условных операторов `if` или операторов цикла `while`), поэтому при трансляции на машинный язык нет никаких ветвлений. Базовые элементы связываются операторами управления.

Программа в такой форме может быть представлена в виде ориентированного графа, как показано на рис. 4.30. Здесь мы вычисляем сумму кубов четных и нечетных целых чисел до какого-либо предела и помещаем результаты в *evensum* и *oddsum* соответственно (листинг 4.6). В пределах каждого базового элемента технологии, упомянутые в предыдущем разделе, работают отлично.

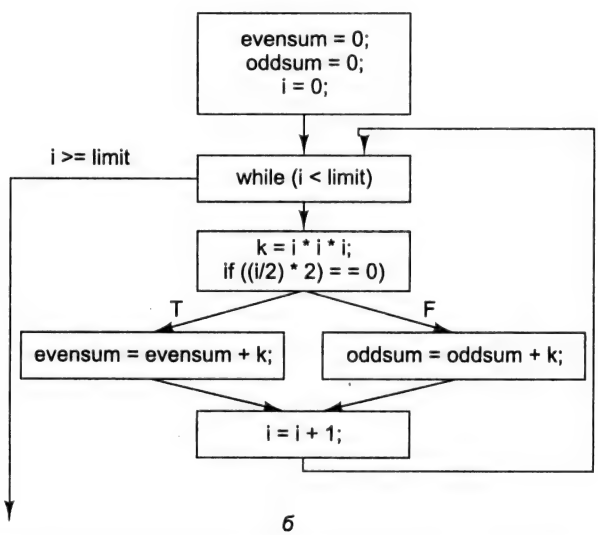
Листинг 4.6. Фрагмент программы

```
evesum=0;
oddsum=0;
i=0;
while (I<limit) {
       k=i*i*i;
       if(((i/2)*2)==i)
           evensum=evensum+k;
       else
           oddsum=oddsum+k;
       i=i+1;
}
```

Проблема состоит в том, что большинство базовых элементов очень короткие и в них недостаточно параллелизма. Следовательно, нужно сделать так, чтобы переупорядочение последовательности команд можно было применять не только в пределах конкретного базового элемента. Выгоднее всего будет передвинуть потенциально медленную операцию в графе повыше, чтобы ее выполнение началось раньше. Это может быть команда LOAD, операция с плавающей точкой или даже начало длинной цепи зависимостей. Перемещение кода вверх по ребру графа называется **подъемом**.

```
evensum = 0;
oddsum = 0;
i = 0;
while (i < limit) {
    k = i * i * i;
    if ((i/2) * 2) == 0)
        evensum = evensum + k;
    else
        oddsum = oddsum + k;
    i = i + 1;
}
```

а



б

Рис. 4.30. Граф базового элемента для фрагмента программы, приведенного в листинге 4.6

Посмотрите на рис. 4.30. Представим, что все переменные были помещены в регистры, кроме *evensum* и *oddsum* (из-за недостатка регистров). Тогда имело бы смысл переместить команды LOAD в начало цикла до вычисления переменной k,

чтобы выполнение этих команд началось раньше, а полученные результаты были бы доступны в тот момент, когда они понадобятся. Естественно, при каждой итерации требуется только одно значение, поэтому остальные команды LOAD будут отбрасываться, но если кэш-память и основная память конвейеризированы, то подобная процедура имеет смысл. Выполнение команды до того, как стало известно, понадобится ли вообще эта команда, называется **спекулятивным выполнением**. Чтобы использовать эту технологию, требуется поддержка компилятора, аппаратного обеспечения, а также некоторое усовершенствование архитектуры. В большинстве случаев переупорядочение команд за пределами одного базового элемента находится вне компетенции аппаратного обеспечения, поэтому компилятор должен перемещать команды явным образом.

В связи со спекулятивным выполнением команд возникают некоторые интересные проблемы. Например, очень важно, чтобы ни одна из спекулятивных команд не имела окончательного результата, который нельзя отменить, поскольку позднее может оказаться, что эти команды не нужно было выполнять. Обратимся к листингу 4.6 и рис. 4.30. Очень удобно производить сложение, как только появляется значение k (даже до условного оператора `if`), но нежелательно сохранять результаты в памяти. Чтобы предотвратить перезапись регистров до того, как стало известно, полезны ли полученные результаты, нужно переименовать (подменить) все выходные регистры, которые используются спекулятивной командой. Как вы можете себе представить, счетчик обращений для отслеживания всего этого очень сложен, но при наличии соответствующего аппаратного обеспечения его вполне можно сделать.

Однако при наличии спекулятивных команд возникает еще одна проблема, которую нельзя решить путем подмены регистров. А что происходит, если спекулятивная команда вызывает исключение (exception)? В качестве примера можно привести команду LOAD, которая вызывает промах кэш-памяти в компьютере с достаточно большим размером строки кэш-памяти (скажем, 256 байт) и памятью, которая работает гораздо медленнее, чем центральный процессор и кэш. Если нам требуется команда LOAD и работа машины останавливается на много циклов, на то время, пока загружается строка кэш-памяти, то это не так страшно, поскольку данное слово действительно нужно. Но если машина простаивает, чтобы вызвать слово, которое, как окажется позднее, совершенно ни к чему, это совершенно не рационально. Если подобных «оптимизаций» слишком много, то центральный процессор будет работать медленнее, чем если бы этих «оптимизаций» вообще не было. (Если машина содержит виртуальную память, которая обсуждается в главе 6, то спекулятивное выполнение команды LOAD может даже вызвать обращение к отсутствующей странице. Подобные ошибки могут сильно повлиять на производительность, поэтому важно их избегать.)

В ряде современных компьютеров данная проблема решается следующим образом. В них содержится специальная команда SPECULATIVE-LOAD, которая производит попытку вызвать слово из кэш-памяти, а если слова там нет, просто прекращает вызов. Если значение находится там и если в данный момент оно действительно требуется, его можно использовать, но если оно в данный момент не требуется, аппаратное обеспечение должно сразу получить это значение. А если окажется, что данное значение нам не нужно, то никаких потерь не будет.

Более сложную ситуацию можно проиллюстрировать следующим выражением:

```
if (x>0) z=y/x;
```

где x, y и z — переменные с плавающей точкой. Предположим, что все эти переменные поступают в регистры заранее и что команда деления с плавающей точкой (эта команда выполняется медленно) перемещается вверх и выполняется еще до условного оператора `if`. К сожалению, если x равен 0, то программа завершается в результате попытки деления на 0. Таким образом, спекулятивная команда приводит к сбою в изначально правильной программе. Еще хуже то, что программист изменяет программу, чтобы предотвратить подобную ситуацию, но сбой все равно происходит.

Одно из возможных решений — специальные версии команд, которые могут вызвать исключения (exceptions). Кроме того, к каждому регистру добавляется специальный бит **(poison bit)**. Если спекулятивная команда дает сбой, она не вызывает trap (ловушку), а устанавливает бит присутствия в регистр результатов. Если этот регистр позднее используется обычной командой, происходит trap (как и должно быть). Однако если этот результат не используется, бит присутствия сбрасывается и не причиняет программе никакого вреда.

Примеры микроархитектурного уровня

В этом разделе мы рассмотрим три современных процессора в свете понятий, изученных в этой главе. Наше изложение будет кратким, поскольку компьютеры чрезвычайно сложны, содержат миллионы вентилей и у нас нет возможности давать подробное описание. Примеры будут те же, которые мы использовали до сих пор: Pentium II, UltraSPARC II и picoJava II.

Микроархитектура процессора Pentium II

Pentium II — один из процессоров семейства Intel. Он поддерживает 32-битные операнды и арифметику, 64-битные операции с плавающей точкой, а также 8- и 16-битные операнды и операции, которые унаследованы от предыдущих процессоров данного семейства. Процессор может адресовать до 64 Гбайт памяти и считывать слова из памяти по 64 бита за раз. Обычная система Pentium II изображена на рис. 3.47.

Как мы уже говорили раньше и как показано на рис. 3.40, картридж с однорядным расположением контактов (SEC) системы Pentium II состоит из двух интегральных схем: центрального процессора (на котором находится разделенная кэш-память первого уровня) и объединенной кэш-памяти второго уровня. На рис. 4.31 показаны основные компоненты центрального процессора: блок вызова/декодирования, блок отправки/выполнения и блок возврата, которые вместе действуют как конвейер высокого уровня. Эти три блока обмениваются данными через пул команд — место для хранения информации о частично выполненных командах. Информация в пуле команд находится в таблице, которая называется **ROB (ReOrder Buffer — буфер переупорядочивания команд)**. Если излагать кратко,

блок вызова/декодирования вызывает команды и разбивает их на микрооперации для хранения в ROB. Блок отправки выполнения получает микрооперации из буфера ROB и выполняет их. Блок возврата завершает выполнение каждой операции и обновляет регистры. Команды поступают в буфер ROB по порядку, могут выполняться в произвольном порядке, но завершаться опять должны по порядку.

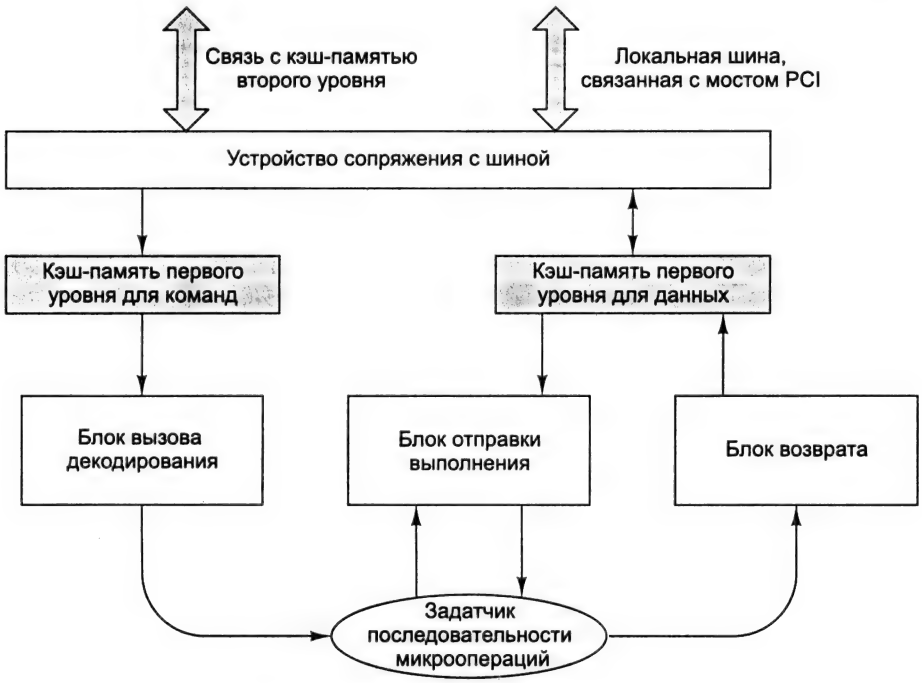


Рис. 4.31. Микроархитектура Pentium II

Блок сопряжения с шиной отвечает за обмен информацией с системой памяти (и с кэш-памятью второго уровня, и с основной памятью). Кэш-память второго уровня не связана с локальной шиной, поэтому блок сопряжения с шиной отвечает за вызов данных из основной памяти через локальную шину, а также за загрузку всех блоков кэш-памяти. Система Pentium II использует протокол синхронизации кэш-памяти MESI, который мы будем рассматривать, когда дойдем до мультипроцессоров в разделе «Архитектуры UMA SMP с шинной организацией» главы 8.

Блок вызова/декодирования

Блок вызова/декодирования отличается высокой степенью конвейеризации (содержит семь стадий). На рис. 4.32 эти семь стадий обозначены IFU0, ..., ROB. Блок отправки/выполнения и блок возврата имеют еще пять стадий, то есть всего стадий 12. Команды поступают на конвейер на стадии IFU0 (IFU — аббревиатура от Instruction Fetch Unit — блок выборки команд), куда из кэша команд загружаются целые 32-байтные строки. Всякий раз, когда внутренний буфер пуст, туда копируется следующая строка кэш-памяти. Регистр NEXT IP (NEXT Instruction Pointer — следующий указатель команды) управляет процессом вызова команд.

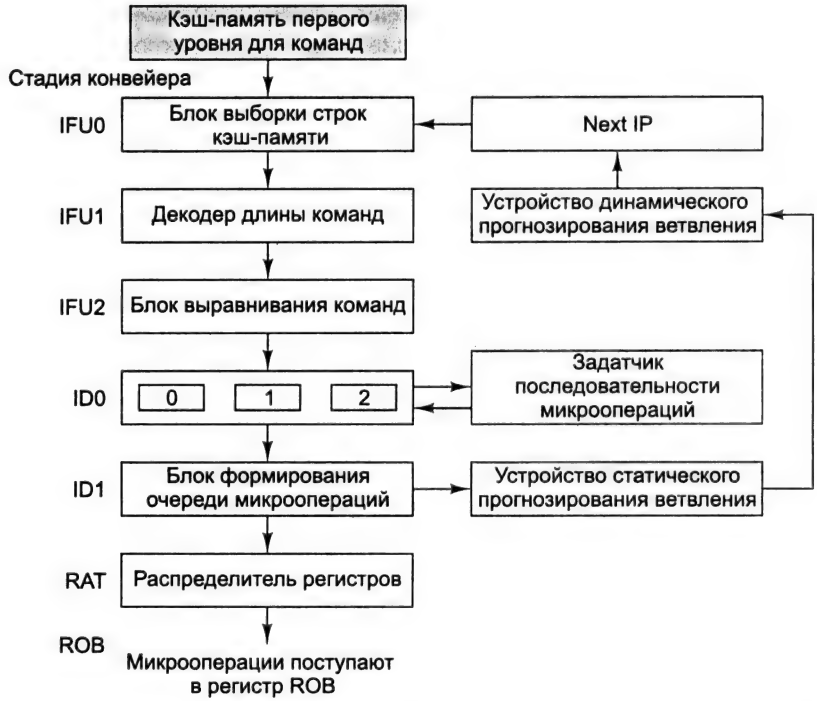


Рис. 4.32. Внутренняя структура блока вызова/декодирования (в упрощенном виде)

Поскольку в наборе команд Intel, который часто называют **IA-32** (32-разрядная архитектура для процессоров Intel), содержатся команды разной длины и различного формата, на следующей стадии, IFU1, происходит анализ потока байтов, чтобы определить начало каждой команды. В случае необходимости на стадии IFU1 может рассматриваться до 30 команд архитектуры IA-32 вперед. К сожалению, вследствие этого обычно встречаются 4 или 5 условных переходов, не все из которых правильно прогнозируются, поэтому в обработке такого большого количества команд заранее нет особого смысла. На стадии IFU2 команды выравниваются, поэтому в следующей стадии они без труда декодируются.

Декодирование начинается на стадии ID0 (Instruction Decoding — декодирование команды). Декодирование в системе Pentium II состоит из превращения каждой команды IA-32 в одну или несколько микроопераций, как и в микроархитектуре Mic-4. Простые команды, например перемещение из одного регистра в другой, переделываются в одну микрооперацию. Выполнение более сложных команд может занимать до четырех микроопераций. Несколько чрезвычайно сложных команд требуют еще больше микроопераций и используют ПЗУ последовательности микроопераций для упорядочения этих микроопераций.

На стадии ID0 имеется три внутренних декодера. Два из них предназначены для простых команд, а третий обрабатывает остальные команды. На выходе получается последовательность микроопераций. Каждая микрооперация содержит код операции, два входных и один выходной регистр.

Очередь микрокоманд выстраивается на стадии ID1. Этот блок аналогичен блоку формирования очереди, изображенному на рис. 4.23. На этой стадии также про-

исходит прогнозирование ветвления (сначала статическое, на всякий случай). Прогноз зависит от нескольких факторов, но для переходов, связанных с текущей командой, считается, что переходы назад будут производиться, а переходы вперед — нет. Затем идет динамическое прогнозирование с использованием специального алгоритма, как показано на рис. 4.29, только в данном случае для прогнозирования используется не два, а четыре бита. Должно быть ясно, что если речь идет о конвейере с 12 стадиями, очень велика вероятность неправильного предсказания, и поэтому нужно так много битов. Если перехода в таблице динамики нет, используется статическое прогнозирование.

Чтобы избежать взаимозависимостей WAR и WAW, система Pentium II поддерживает переименования (подмены), как мы видели в табл. 4.13. Реальные регистры в командах IA-32 могут быть заменены в микрооперациях любым из 40 внутренних временных регистров, находящихся в буфере ROB. Подмена происходит на стадии RAT.

И наконец, микрооперации копируются в буфер ROB со скоростью три микрооперации за цикл. Сюда же собираются операнды, если они имеются в наличии. Если операнды микрооперации и регистр результатов доступны, а операционный блок свободен, микрооперацию можно выпустить. В противном случае она находится в буфере ROB, пока не появятся все необходимые ресурсы.

Блок отправки/выполнения

Перейдем к блоку отправки/выполнения, который изображен на рис. 4.33. Этот блок устанавливает очередность и выполняет микрооперации, разрешает взаимозависимости и конфликты ресурсов. Хотя за один цикл можно декодировать всего три команды (на стадии ID0), за один цикл можно выпустить для выполнения целых пять микроопераций, по одной на каждый порт. Такую скорость нельзя поддерживать, поскольку она превышает способности работы блока возврата. Микрооперации могут запускаться не по порядку, но блок возврата должен завершать их выполнение по порядку. Чтобы следить за микрооперациями, регистрами и функциональными блоками, требуется сложный счетчик обращений. Когда операция готова для выполнения, она может начаться, даже если другие операции, которые поступили в буфер ROB раньше нее, еще не готовы. Если несколько микроопераций пригодны для выполнения одним и тем же функциональным блоком, с помощью сложного алгоритма выбирается важнейшая из них, и именно она и запускается следующей. Например, выполнение перехода гораздо важнее, чем выполнение арифметического действия, поскольку первый из них влияет на ход программы.

Блок отправки/выполнения состоит из резервации и функциональных блоков, которые связаны с пятью портами. Резервация представляет собой очередь из 20 элементов для микроопераций, которые имеют собственные операнды. Они ожидают своей очереди в резервации, пока не освободится нужный функциональный блок.

Между резервацией и функциональными блоками имеется пять портов. Некоторые функциональные блоки разделяют один порт, как показано на рисунке. Блок загрузки и блоки запоминающих устройств выдают информацию для операций загрузки и сохранения соответственно. Для запоминающих устройств есть два порта. Поскольку через один порт за один цикл может выдаваться только одна микрооперация, то если двум микрооперациям нужно пройти через один и тот же порт, одной из них придется подождать.

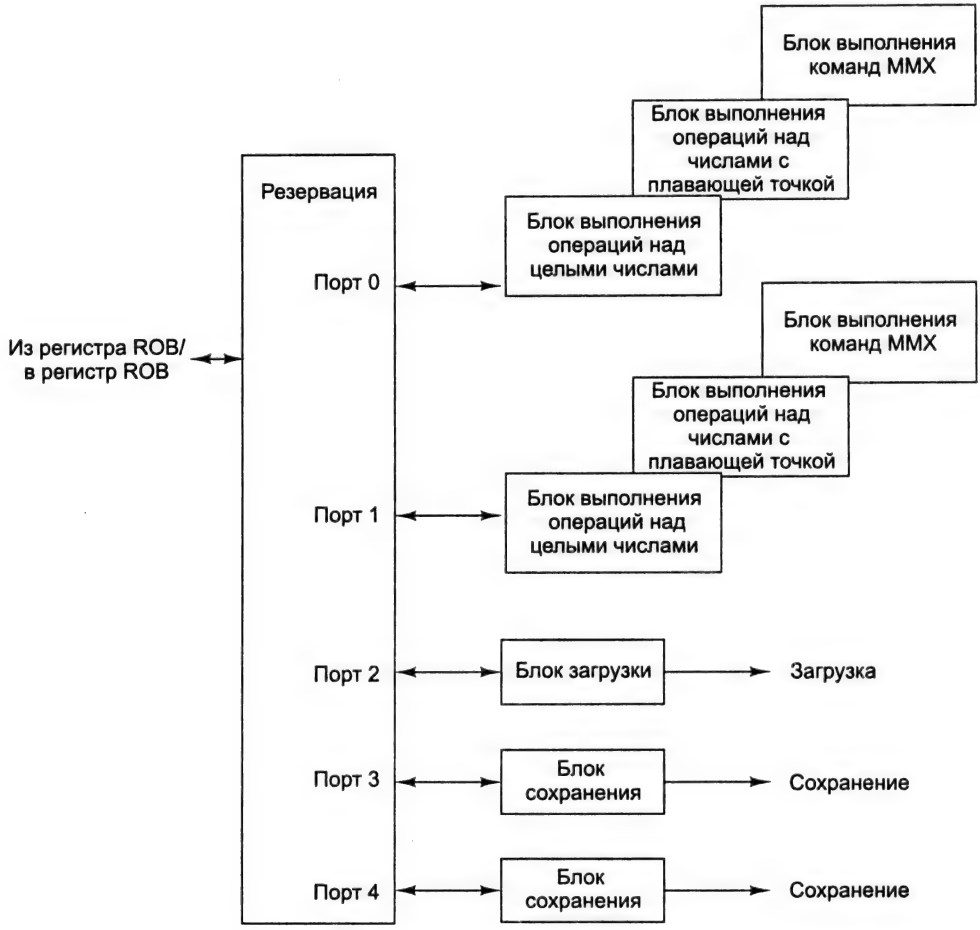


Рис. 4.33. Блок отправки/выполнения

Блок возврата

Когда микрооперация выполнена, она переходит обратно в резервацию, а затем в буфер ROB, и там ожидает возврата. Блок возврата отвечает за отправку результатов в нужные места — в соответствующий регистр или в другие устройства блока отправки/выполнения, которым требуется данное значение. Блок отправки/выполнения содержит «официальные» регистры, то есть те, в которых хранятся значения завершенных команд. Блок возврата содержит ряд «промежуточных» регистров, значения которых были вычислены командой, которая еще не завершилась, поскольку выполнение предыдущих команд не закончилось.

Система Pentium II поддерживает процедуру спекулятивного выполнения, поэтому некоторые команды будут выполняться напрасно, и их результаты никуда не нужно будет сбрасывать. Именно поэтому и нужна способность возвращаться в предыдущее состояние. Если стало известно, что какая-то микрооперация пришла из команды, которую не нужно было выполнять, результаты этой микрооперации

отбрасываются. Все это контролирует блок возврата. Только результаты «официально» выполненных команд могут возвращаться в регистры, причем это должно происходить в том же порядке, что и в программе, даже если команды выполнялись в произвольном порядке.

Микроархитектура процессора UltraSPARC II

Серия UltraSPARC, произведенная компанией Sun, — это реализация версии 9 архитектуры SPARC. Все модели сходны друг с другом и различаются главным образом производительностью и ценой. Тем не менее, чтобы избежать путаницы, в этом разделе мы будем говорить о системе UltraSPARC II и описывать те характеристики, по которым этот процессор отличается от других процессоров того же семейства.

UltraSPARC II — это 64-разрядная машина с 64-разрядными регистрами и 64-разрядным трактом данных, но в целях совместимости с машинами версии 8 (которые являются 32-разрядными) она может обращаться с 32-разрядными операндами, а программное обеспечение, написанное для 32-разрядных версий SPARC, изменять не нужно. Хотя внутренняя архитектура машины использует 64 разряда, ширина шины памяти составляет 128 битов, аналогично процессору Pentium II с 32-разрядной архитектурой и 64-разрядной шиной памяти. Ядро системы UltraSPARC II показано на рис. 3.44.

Вся серия SPARC с самого начала представляла собой систему RISC. У большинства команд есть два входных и один выходной регистр, поэтому они хорошо подходят для конвейерного выполнения в одном цикле. Разбивать старые команды CISC на микрооперации RISC, как в системе Pentium II, не нужно.

UltraSPARC II — это суперскалярная машина, которая может выдавать 4 команды за цикл. Команды запускаются по порядку, но завершаться могут и в произвольном порядке. Тем не менее прерывания являются точными (то есть всегда точно известно, в каком месте программы была машина, когда произошло прерывание). Существует аппаратная поддержка для спекулятивных загрузок в виде команды PREFETCH, которая не вызывает ошибок при промахе кэша. Она даже не блокирует последовательные обращения к памяти. Следовательно, компилятор может вставлять одну или несколько команд PREFETCH задолго до того, как они понадобятся, в надежде, что они пригодятся позже, не испытывая никаких неприятностей в случае, если нужного слово не окажется в кэш-памяти.

Общий обзор системы UltraSPARC II

Диаграмма UltraSPARC II представлена на рис. 4.34. Все указанные компоненты расположены на микросхеме центрального процессора. Исключение составляет кэш-память второго уровня, которая является внешней по отношению к процессору. Кэш команд — это 16 Кбайт двувходовой ассоциативной кэш-памяти, со строками по 32 байта и с возможностью возврата половины строки. Половина строки кэш-памяти (16 байтов) содержит ровно четыре команды, и все эти четыре команды могут выдаваться за один цикл. Кэш-память данных — это 16 Кбайт кэш-памяти прямого отображения со сквозной записью и без заполнения по записи. Здесь тоже используются 32-байтные строки, разделенные на 2 части по 16 байтов.

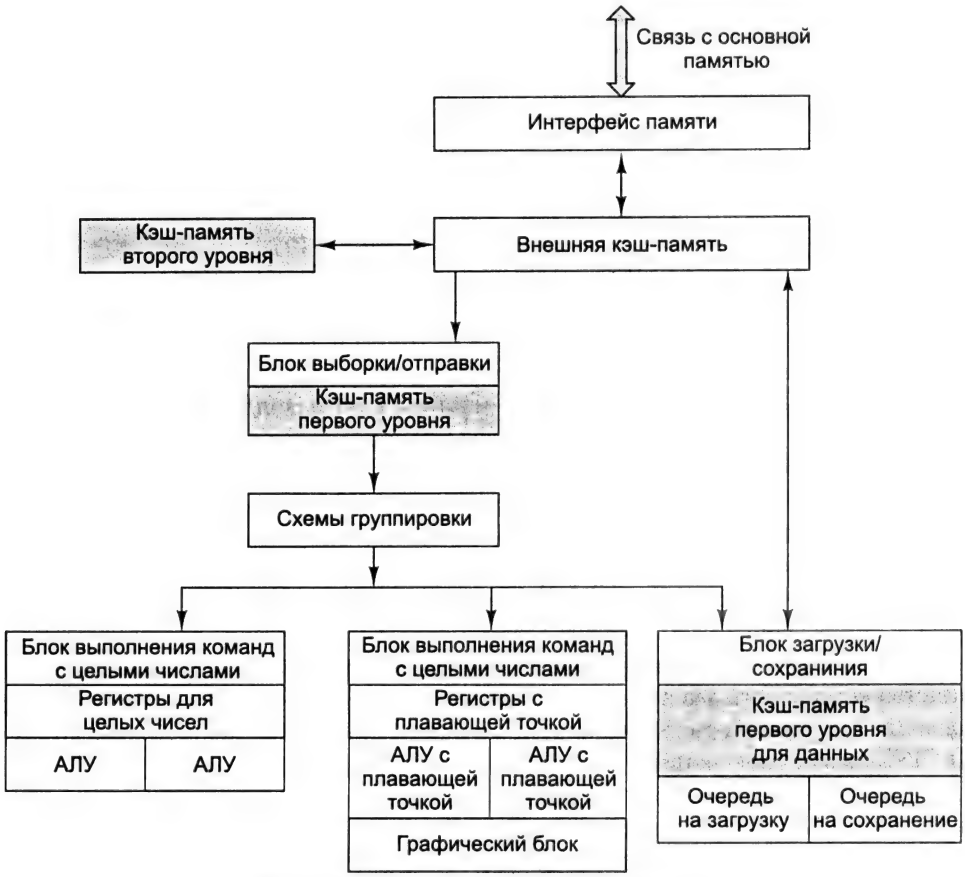


Рис. 4.34. Микроархитектура UltraSPARC II

В случае промаха кэш-памяти первого уровня нужная строка ищется в кэш-памяти второго уровня. Если поиск завершился успехом, строка копируется в кэш-память первого уровня. В случае неудачи внешняя кэш-память (кэш-память второго уровня) посылает устройству сопряжения с памятью команду вызова строки из основной памяти.

Рассмотрим функциональные блоки системы UltraSPARC II. Блок выборки отправки в целом похож на блок вызова/декодирования в системе Pentium II (см. рис. 4.31). Однако у этого блока работа проще, поскольку входные команды уже представлены в трех регистрах в виде микроопераций, поэтому их не нужно разбивать. Блок может вызывать команды и из кэш-памяти первого уровня, и из кэш-памяти второго уровня без потери времени. За один цикл вызываются четыре команды.

Чтобы сократить неприятные последствия неправильно предсказанных переходов, каждая группа из четырех команд в кэш-памяти команд содержит адрес, который указывает, какую именно половинчатую строку нужно взять следующей. Кроме того, предсказывающее устройство находится внутри блока выборки отправки.

При этом используется 2-битный алгоритм прогнозирования, сходный с тем, который показан на рис. 4.29. Более того, UltraSPARC II содержит ряд команд перехода, в которых компилятор может сообщать аппаратному обеспечению, каким именно способом предсказывать переход. Вызванные заранее команды помещаются в очередь из 12 элементов, а затем передаются в схему группировки.

Схема группировки — это блок, который выбирает по четыре команды за один раз из очереди для запуска. Задача состоит в том, чтобы найти 4 команды, которые можно выпустить одновременно. Блок целых чисел содержит два раздельных АЛУ, что позволяет выполнять две команды параллельно. Блок вычислений с плавающей точкой также содержит два АЛУ. Следовательно, в одной группе может находиться по две команды каждого типа, но не четыре команды одного типа. Чтобы сделать группирование оптимальным, команды должны запускаться не по порядку, что разрешается. Завершаются они также в произвольном порядке.

Блок целых чисел и блок вычислений с плавающей точкой содержат собственные регистры, поэтому команды берут операнды прямо внутри блока и там же оставляют результаты. Регистр целых чисел и регистр с плавающей точкой разделены, поэтому значения никогда не переходят из одного блока в другой. В блоке вычислений с плавающей точкой также находится графический блок, который выполняет специальные команды для двух- и трехмерных изображений, аудио и видео, аналогичные командам MMX в системе Pentium II.

Блок загрузки/сохранения управляет командами LOAD и STORE. Если они имеются в кэш-памяти данных первого уровня, то выполняются без задержки. В противном случае нужная строка берется из кэш-памяти второго уровня или основной памяти (если речь идет о команде LOAD) или нужное слово записывается туда (если речь идет о команде STORE). Если бы кэш-память данных была write-allocate, тогда в случае промаха кэш-памяти при записи (команда STORE) нужная строка переносилась бы из кэш-памяти второго уровня или из основной памяти. На самом деле, если нужно сохранить отдельное слово, просто осуществляется сквозная запись в кэш-память второго уровня, а в случае промаха — в основную память. Чтобы избежать блокирования из-за отсутствия нужного слова в кэш-памяти данных, блок загрузки/сохранения хранит очереди незавершенных команд LOAD и STORE, поэтому можно продолжать обрабатывать новые команды, пока завершается выполнение старых.

Конвейеризация системы UltraSPARC II

Система UltraSPARC II содержит конвейер с 9 стадиями. Некоторые из этих стадий различны для команд с целыми числами и команд с плавающей точкой (рис. 4.35). На первой стадии вызываются команды из кэш-памяти команд (если это возможно). При благоприятных обстоятельствах (отсутствии промахов кэш-памяти, неправильного прогнозирования ветвлений, сложных команд, наличии правильной смеси команд и т. п.) машина может продолжать вызывать и запускать по 4 команды за цикл. На стадии декодирования перед копированием команд в очередь к каждой команде прибавляются дополнительные биты. Эти биты ускоряют последующую обработку (например, сразу отправляя команду в соответствующий функциональный блок).

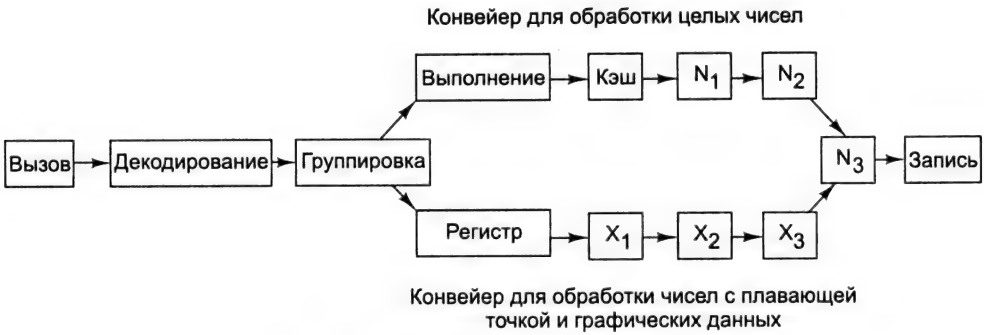


Рис. 4.35. Конвейер системы UltraSPARC II

Стадия группировки соответствует схеме группировки, которую мы рассматривали раньше. На этой стадии декодированные команды объединяются в группы. В каждой группе находится по 4 команды. Все команды одной группы должны быть подобраны таким образом, чтобы их можно было выполнять одновременно.

С этого момента стадии конвейера для операций с целыми числами и для операций с плавающей точкой разделяются. На стадии 4 в блоке целых чисел большинство команд выполняется прямо за один цикл. Однако команды STORE и LOAD требуют дополнительной обработки на стадии кэширования. На стадиях N_1 и N_2 не производится никаких действий для команд, но эти стадии нужны для синхронизации работы двух конвейеров. Если каждая команда с целыми числами будет завершаться на несколько секунд позже, это не такая уж большая потеря, зато конвейер работает равномерно.

Блок с плавающей точкой содержит отдельные 4 стадии. Первая нужна для доступа к регистрам с плавающей точкой. Следующие три нужны для выполнения команды. Все команды с плавающей точкой выполняются за три цикла, за исключением деления (на эту операцию требуется 12 циклов) и квадратного корня (здесь нужно 22 цикла), поэтому длинная последовательность других команд не снижает скорости работы конвейера.

Стадия N_3, общая для обоих блоков, нужна для разрешения исключительных ситуаций, например деления на нуль. Наконец, на последней стадии результаты записываются обратно в регистры. Эта стадия напоминает блок возврата системы Pentium II в том, что если команда прошла через эту стадию, она завершена.

Микроархитектура процессора picoJava II

В системе picoJava II двоичные программы JVM могут работать практически без интерпретации. Большинство команд JVM выполняются непосредственно аппаратным обеспечением за один цикл. Около 30 команд JVM являются микропрограммными. Только очень небольшое число команд не может выполняться аппаратным обеспечением picoJava II и вызывает traps (ловушки). Эти команды связаны с особенностями JVM, которые мы не обсуждали, например создание и управление сложными программными объектами.

Общий обзор системы picoJava II

Диаграмма микроархитектуры picoJava II представлена на рис. 4.36. Микросхема процессора содержит разделенную кэш-память первого уровня. Кэш-память команд факультативна. Ее объем может составлять 1 Кбайт, 2 Кбайт, 4 Кбайт, 8 Кбайт или 16 Кбайт. Это кэш-память прямого отображения. Размер строки составляет 16 байтов. Кэш-память данных тоже факультативна, и ее объем может составлять 1 Кбайт, 2 Кбайт, 4 Кбайт, 8 Кбайт или 16 Кбайт. Это двувходовая ассоциативная кэш-память. Размер строки также составляет 16 байтов. Она использует обратную запись и заполнение по записи. Каждая кэш-память соединяется с шиной памяти по 32-битному каналу. Система microJava 701 имеет оба блока кэш-памяти в обязательном порядке, объем каждого из них составляет 16 Кбайт. Факультативный блок с плавающей точкой также является частью разработки picoJava II.

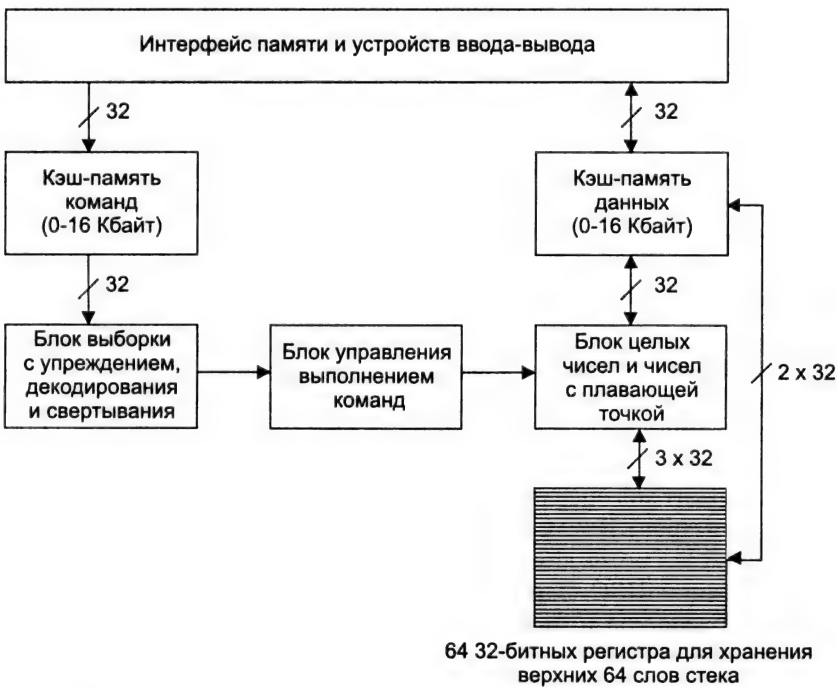


Рис. 4.36. Диаграмма системы picoJava II с кэш-памятью первого уровня и блоком с плавающей точкой. Это конфигурация системы microJava 701

Кэш-память команд передает в блок вызова, декодирования и свертывания по 8 байтов за раз. Этот блок, в свою очередь, связан с контроллером выполнения и с основным трактом данных (блоком операций с целыми числами и с плавающей точкой). Ширина тракта данных составляет 32 бита для целочисленных операций. Этот тракт данных может также управляться с плавающей точкой с одинарной и двойной точностью (IEEE 754).

Наиболее интересная часть рис. 4.36 — это регистровый файл, состоящий из 64 32-битных регистров. В этих регистрах могут содержаться верхние 64 слова стека JVM, что сильно повышает скорость доступа к словам в стеке. И стек операндов,

и стек локальных переменных под ним могут находиться в регистровом файле. Доступ к регистровому файлу «свободный» (то есть происходит без задержек, тогда как доступ к кэш-памяти данных требует дополнительного цикла). Ширина канала между регистровым файлом и блоком операций с целыми числами и с плавающей точкой составляет 96 битов. За один цикл канал выдерживает 2 32-битных считывания из стека и одну 32-битную запись в стек.

Если, например, стек операндов состоит из двух слов, то в регистровом файле может находиться до 62 слов локальных переменных. Естественно, при помещении еще одного слова в стек возникает проблема. Происходит так называемый **дрибблинг** — это когда одно или несколько слов, находящихся глубоко в стеке, записываются обратно в память. Точно так же, если несколько слов выталкиваются из стека операндов, в регистровом файле освобождается место, и поэтому некоторые слова, находящиеся глубоко в стеке, могут перезагружаться в регистровый файл. Специальные регистры на микросхеме определяют, насколько полным должен быть регистр, чтобы слова из нижней части стека записывались в память, и насколько пустым он может быть для того, чтобы перезагрузить регистровый файл из памяти. Чтобы легко произвести дрибблинг без копирования, регистровый файл действует как кольцевой буфер с указателями на самое нижнее и на самое верхнее слова. Дрибблинг происходит автоматически всякий раз, когда регистровый файл переполняется или пустеет.

Конвейер системы picoJava II

Конвейер системы picoJava II состоит из шести стадий. Он показан на рис. 4.37. На первой стадии из кэш-памяти команд в буфер команд вызываются команды по 8 байтов за раз. Емкость буфера команд составляет 16 байтов. На следующей стадии команды декодируются и определенным образом объединяются. На выходе из блока декодирования получается последовательность микроопераций, каждая из которых содержит код операции и три номера регистров (двух входных и одного выходного регистров). В этом отношении машина picoJava II сходна с Pentium II: обе машины получают поток команд CISC, который превращается в последовательность микроопераций RISC. Однако, в отличие от Pentium II, машина picoJava II не является суперскалярной и микрооперации выполняются и завершаются в том порядке, в котором они запускаются. В случае промаха кэш-памяти, если операнд приходится вызывать из основной памяти, процессор должен простаивать.

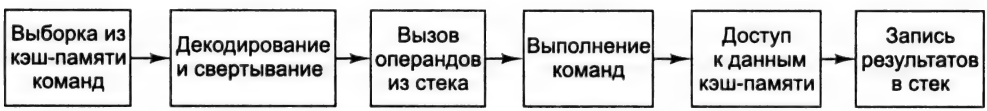


Рис. 4.37. Шесть стадий конвейера в машине picoJava II

На третьей стадии вызываются операнды из стека (фактически из регистрового файла), чтобы они были в наличии для четвертой стадии. Четвертая стадия — это блок выполнения команд. На пятой стадии в случае необходимости производится обращение к кэш-памяти данных (например, чтобы сохранить там результаты). Наконец, на шестой стадии результаты записываются обратно в стек.

Свертывание команд

Как мы упоминали выше, блок декодирования способен **свертывать** команды вместе. Чтобы объяснить, как происходит этот процесс, рассмотрим следующее выражение:

```
n=k+m;
```

Трансляция на (I)JVM может быть следующей:

```
ILOAD 7
ILOAD 1
IADD
   ISTORE 3,
```

Предполагается, что k, m и n — локальные переменные 7, 1 и 3 соответственно. Процесс выполнения этих четырех команд изображен на рис. 4.38, *а*.

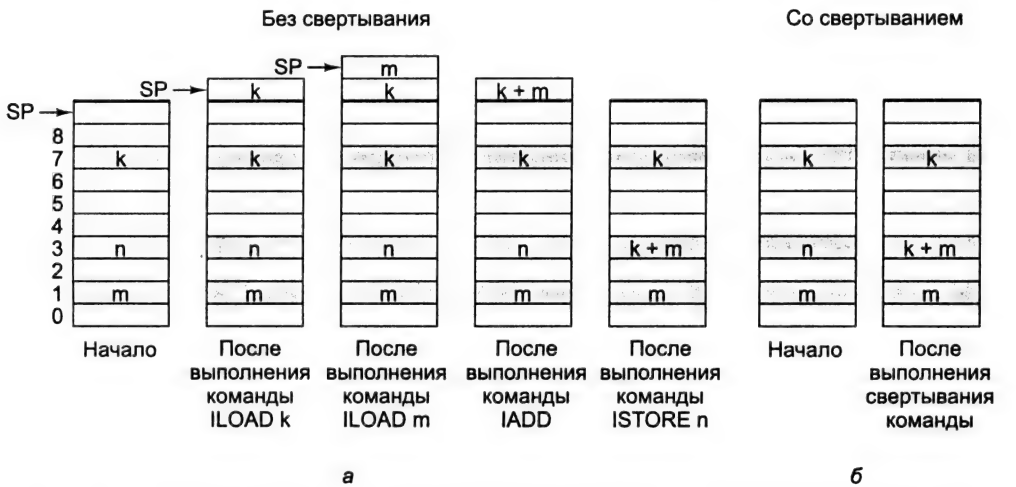


Рис. 4.38. Выполнение последовательности из четырех команд для вычисления выражения n=k+m (*а*); та же последовательность, свернутая до одной команды (*б*)

Если предположить, что все три переменные находятся достаточно высоко в стеке, настолько высоко, что все они содержатся в регистровом файле, то для выполнения этой последовательности команд вообще не требуются обращения к памяти. Первая команда `ILOAD 7` копирует слово, находящееся в седьмой локальной переменной, в вершину стека и увеличивает указатель стека на 1. Сходным образом команда `ILOAD 1` производит копирование из регистра в регистр. Команда `IADD` складывает два регистра, а команда `ISTORE` копирует значение регистра в регистровый файл. Избавление от любых обращений к памяти — главный способ улучшения производительности.

Однако машина picoJava II делает не только это. Данная последовательность из четырех команд просто складывает два регистра и сохраняет полученное значение в третьем регистре. Блок декодирования определяет это условие и запускает одну микрооперацию: трехрегистровую команду `ADD`. Таким образом, вместо четырех команд JVM, для которых требуется 9 обращений к памяти, мы получаем одну

микрооперацию для сложения, как показано на рис. 4.38, *б*. И хотя на входе в конвейер были команды CISC с многочисленными обращениями к памяти, в результате выполнена была всего одна простая микрооперация. Таким образом, picoJava II может выполнять программы на языке Java, скомпилированные для JVM, так же быстро, как будто они были скомпилированы на машинный язык компьютера RISC.

Как мы только что увидели, возможность сворачивать несколько команд JVM в одну микрооперацию является ключом к высокой производительности. Следовательно, стоит кратко изложить, как блок декодирования осуществляет свертывание. Для этого команды распределяются по шести группам (табл. 4.14). Первая группа содержит команды, которые не сворачиваются. Во второй находятся команды загрузки локальных переменных. В IJVM имеется одна такая команда, `ILOAD`, а JVM содержит и другие команды. Третья группа состоит из команд запоминания, например `ISTORE`. Четвертая и пятая группы предназначены для команд переходов с одним и двумя операндами соответственно. Последняя группа состоит из команд, которые выталкивают два операнда из стека, выполняют с ними какие-нибудь вычисления и помещают результат обратно в стек.

Таблица 4.14. Распределение команд JVM по группам для свертывания

Группа	Описание	Пример
NF	Несвертываемые команды	GOTO
LV	Помещают слово в стек	ILOAD
MEM	Выталкивают слово из стека	ISTORE
BG1	Операции с использованием одного стекового операнда	IFEQ
BG2	Операции с использованием двух стековых операндов	IF_CMPEQ
OP	Вычисления над двумя операндами с одним результатом	IADD

Блок декодирования передает 74-битные микрооперации операционному блоку (через блок вызова операндов). Большинство этих микроопераций содержат код операции и три регистра и могут быть выполнены за один цикл. Когда блок декодирования выталкивает команды JVM из буфера команд, он превращает их в последовательность микроопераций. Кроме того, блок декодирования определяет, какие последовательности команд JVM можно свернуть в одну микрооперацию. В такой последовательности может быть до четырех команд. Если обнаружена подходящая последовательность, выдается соответствующая микрооперация, а изначальные команды отбрасываются.

В табл. 4.15 приведены некоторые типичные последовательности команд JVM, которые можно свернуть. Когда в блоке декодирования оказывается одна из таких последовательностей, он замещает обычное разбиение команд на одну микрооперацию, которая выполняет работу всей этой последовательности за один цикл. Например, он превращает последовательность из четырех команд в одну трехрегистровую микрооперацию `ADD`, как мы видели на рис. 4.38. Процесс свертывания происходит только тогда, когда требуемые локальные переменные находятся достаточно близко от вершины стека, то есть содержатся в регистровом файле.

Таблица 4.15. Некоторые последовательности команд JVM, которые можно сворачивать

Последовательность команд				Пример
LV	LV	OP	MEM	ILOAD, ILOAD, IADD, ISTORE
LV	LV	OP		ILOAD, ILOAD, IADD
LV	LV	BG2		ILOAD, ILOAD, IF_CMPEQ
LV	BG1			ILOAD, IFEQ
LV	BG2			ILOAD, IF_CMPEQ
LV	MEM			ILOAD, ISTORE
OP	MEM			IADD, ISTORE

Свертывание команд происходит довольно часто, поэтому существенная часть программы JVM может быть выполнена настолько быстро, как будто она была скомпилирована прямо для конвейеризированного процессора RISC. Измерения показывают, что picoJava II может выполнять программы на языке Java в пять раз быстрее, чем если те же программы скомпилировать на машинный язык для Pentium, который работает с такой же тактовой частотой, и в 15 раз быстрее, чем при интерпретируемом выполнении той же программы на машине Pentium.

В машине picoJava II используется чрезвычайно примитивный алгоритм прогнозирования ветвлений: она всегда предсказывает, что перехода не будет. За этим стоит идея сохранить микросхему простой и дешевой, а не тратить существенное пространство микросхемы на схемы прогнозирования. Однако благодаря длине конвейера (6 стадий вместо 12, как в системе Pentium II) проигрыш при неправильном предсказании перехода составляет всего три цикла.

Сравнение Pentium, UltraSPARC и picoJava

Данные три примера во многом отличаются друг от друга, однако у них есть удивительная общность, которая может сказать кое-что о том, как лучше разрабатывать компьютер. Машина Pentium II содержит старый набор команд CISC, который инженеры компании Intel были бы рады выкинуть в бухту Сан-Франциско, но тогда они нарушили бы законы о загрязнении воды. UltraSPARC II — система RISC. PicoJava II — машина со стековой организацией и командами различной длины, которые совершают огромное число обращений к памяти.

Несмотря на эти различия, все три машины имеют сходные функциональные блоки. Все функциональные блоки принимают микрооперации, которые содержат код операции, два входных регистра и один выходной регистр. Все они могут выполнять микрооперацию за один цикл. Все они конвейеризированы и применяют прогнозирование ветвления. Все они содержат разделенную кэш-память для команд и для данных, объем каждой составляет 16 Кбайт.

Такое внутреннее сходство не случайно, и причиной его является вовсе не постоянные переходы с одной работы на другую инженеров Силиконовой долины. Когда мы рассматривали микроархитектуры Mic-3 и Mic-4, мы увидели, что достаточно просто построить конвейеризированный тракт данных, который исполь-

зует два регистра в качестве источников, пропускает значения этих регистров через АЛУ и сохраняет результат в регистре. На рисунке 4.22 представлено графическое изображение такого конвейера. Для современной техники это наиболее эффективная разработка.

Главное различие между Pentium II, UltraSPARC II и picoJava II — переход от набора команд к функциональному блоку. Компьютеру Pentium II приходится разбивать команды CISC, чтобы переделать их в трехрегистровый формат, который нужен для функционального блока. Именно этот процесс показан на рис. 4.32 — разбиение больших команд на маленькие микрооперации. У машины picoJava II обратная проблема — как скомбинировать несколько команд вместе, чтобы получить простую микрооперацию. Такой процесс называется свертыванием. Машине UltraSPARC II вообще не нужно ничего делать, поскольку ее первоначальные команды уже представляют собой маленькие удобные микрооперации. Вот почему большинство новых архитектур команд — архитектуры типа RISC, если, конечно, у них нет какого-нибудь скрытого мотива (например, вызов программ на языке Java через Интернет и их выполнение на произвольной машине).

Полезно будет сравнить нашу последнюю разработку, микроархитектуру Mic-4, с этими тремя реальными машинами. Mic-4 больше всего похожа на Pentium II. Обе системы интерпретируют команды, которые не являются командами типа RISC. Для этого обе системы разбивают команды на микрооперации с кодом операции, двумя входными регистрами и одним выходным регистром. В обоих случаях помещаются в очередь для дальнейшего выполнения. Микроархитектура Mic-4 запускает микрооперации строго по порядку, выполняет их строго по порядку и завершает выполнение тоже строго по порядку, а Pentium II запускает по порядку, выполняет в произвольном порядке, а завершает опять по порядку. Кроме того, внутренняя структура конвейера Pentium II, особенно та его часть, которая показана на рис. 4.32, во многом сходна с Mic-4.

А теперь сравним Mic-4 с picoJava II. Хотя кажется, что эти две архитектуры должны быть похожи по логике вещей — они интерпретируют один и тот же набор команд, но на самом деле это не совсем так. Причина этого различия состоит в том, что блоки декодирования имеют диаметрально противоположные стратегии. Mic-4 берет каждую входящую команду IJVM и сразу разбивает ее на микрооперации, а picoJava II пытается соединить (свернуть) несколько команд IJVM в одну микрооперацию. Простая операция присваивания `i=j+k` занимает 14 циклов на Mic-4 и 1 цикл на picoJava II. В данном случае свертывание улучшает производительность в 14 раз. Очевидно, что второй метод ведет к более быстрому выполнению команд, но сложность процесса свертывания тоже существенна.

Mic-4 и UltraSPARC II вообще нельзя сравнивать, поскольку команды системы UltraSPARC II — это команды RISC (то есть трехрегистровые микрооперации). Их не нужно ни разбивать, ни объединять. Их можно выполнять в том виде, в каком они есть, каждую за один цикл тракта данных.

Все четыре машины конвейеризированы. Pentium II имеет 12 стадий, UltraSPARC II — 9 стадий, picoJava II — 6 стадий, Mic-4 — 7 стадий. У Pentium II больше стадий, поскольку этой машине приходится разбивать сложные команды. UltraSPARC II содержит больше стадий, чем ему нужно, поскольку конвейер для целочисленных вычислений был искусственно удлинен на 2 стадии, чтобы опера-

ции с целыми числами занимали столько же времени, сколько занимают операции с плавающей точкой. Отсюда следует вывод: при современном состоянии техники оптимальным является конвейер с шестью или семью стадиями, который обрабатывает трехрегистровые микрооперации. Микроархитектура Mic-4 дает хорошее представление о том, как работает такой конвейер (по крайней мере, с командами, которые не являются командами типа RISC).

Краткое содержание главы

Основным компонентом любого компьютера является тракт данных. Он содержит несколько регистров, две или три шины, один или несколько функциональных блоков, например АЛУ, и схему сдвига. Основной цикл состоит из вызова нескольких операндов из регистров и их передачи по шинам к АЛУ и другому функциональному блоку. После выполнения операции результаты сохраняются опять в регистрах.

Тракт данных может управляться задатчиком последовательности, который вызывает микрокоманды из управляющей памяти. Каждая микрокоманда содержит биты, управляющие трактом данных в течение одного цикла. Эти биты определяют, какие операнды нужно выбирать, какую операцию нужно выполнять и что нужно делать с результатами. Кроме того, каждая микрокоманда определяет своего последователя (обычно в ней содержится адрес следующей микрокоманды). Некоторые микрокоманды изменяют этот базовый адрес с помощью операции ИЛИ.

IJVM — это машина со стековой организацией и с 1-байтными кодами операций, которые помещают слова в стек, выталкивают слова из стека и выполняют различные операции над словами из стека (например, складывают их). В главе приводится микропрограмма для микроархитектуры Mic-1. Если добавить блок выборки команд для загрузки команд из потока байтов, то можно устранить большое количество обращений к счетчику команд, и тогда скорость работы машины сильно повысится.

Существует множество способов разработки микроархитектурного уровня. Есть много различных вариантов: двухшинная архитектура — трехшинная архитектура, кодированные поля микрокоманды — декодированные поля микрокоманды, наличие или отсутствие вызова с упреждением и многие другие. Mic-1 — это простая машина с программным управлением, последовательным выполнением команд и полным отсутствием параллелизма. Mic-4, напротив, является высокопараллельной микроархитектурой с конвейером с семью стадиями.

Производительность компьютера можно повысить несколькими способами. Главный способ — использование кэш-памяти. Кэш-память прямого отображения и ассоциативная кэш-память с множественным доступом широко используются для того, чтобы ускорить обращения к памяти. Кроме того, применяется прогнозирование ветвления (как статическое, так и динамическое), исполнение с изменением последовательности и спекулятивное выполнение команд.

Наши три примера, Pentium II, UltraSPARC II и picoJava II, во многом отличаются друг от друга, но при этом удивительно похожи в плане выполнения команд.

Pentium II берет команды CISC и разбивает их на микрооперации, которые обрабатываются суперскалярной архитектурой с прогнозированием ветвления, изменением последовательности команд и спекулятивным выполнением. UltraSPARC II — это современный 64-разрядный процессор с командами типа RISC. Здесь тоже используется прогнозирование ветвления, исполнение с изменением последовательности и спекулятивные команды. PicoJava II представляет собой более простой процессор, предназначенный для дешевых устройств, поэтому у него нет таких особенностей, как динамическое прогнозирование ветвления. Однако при применении свертывания команд эта машина способна выполнять команды JVM достаточно быстро, как будто это регистровые команды RISC. Все три машины содержат сходные функциональные блоки, которые обрабатывают трехрегистровые микрооперации, когда они проходят через конвейер.

Вопросы и задания

1. В табл. 4.1 показан один из способов получения результата *A* на выходе из АЛУ. Приведите другой способ.
2. В микроархитектуре Mic-1 требуется 1 нс на установку регистра MIR, 1 нс — на передачу значения регистра на шину B, 3 нс — на запуск АЛУ и схемы сдвига и 1 нс — на передачу результатов обратно в регистры. Длительность синхронизирующего импульса составляет 2 нс. Может ли такая машина работать с частотой 100 МГц? А 150 МГц?
3. На рис. 4.5 регистр шины B закодирован в 4-битном поле, а шина C представлена в виде битового отображения. Почему?
4. На рис. 4.5 есть блок «Старший бит». Нарисуйте его схему.
5. Когда в микрокоманде установлено поле JMPC, регистр MBR соединяется операцией ИЛИ с полем NEXT_ADDRESS, чтобы получить адрес следующей микрокоманды. Существуют ли такие обстоятельства, при которых имеет смысл использовать JMPC, если NEXT_ADDRESS — 0x1FF?
6. Предположим, что в примере, приведенном в листинге 4.1, выражение `i=0;` добавляется после условного оператора. Каким будет новый код ассемблера? Предполагается, что компилятор является оптимизирующим.
7. Напишите две трансляции IJVM для следующего высказывания на языке Java: `i=j+k+4;`
8. Напишите на языке Java выражение, которое произвело следующую программу IJVM:

```
ILOAD j
ILOAD k
ISUB
BIPUSH 6
ISUB
DUP
IADD
ISTORE i
```

9. В этой главе мы упомянули, что во время трансляции выражения

   ```
   if (Z) goto L1; else goto L2
   ```

   в двоичную форму *L2* должно находиться среди младших 256 слов управляющей памяти. А возможно ли иметь *L1*, скажем, в ячейке с адресом 0x40, а *L2* — в ячейке с адресом 0x140? Объясните, почему.
10. В микропрограмме для Mic-1 в микрокоманде `if_cmpeq3` значение регистра MDR копируется в регистр H, а в следующей строке от него отнимается значение регистра TOS. Казалось бы, это удобнее записать в одном высказывании:

    ```
    if_cmpeq3  Z=MDR-TOS; rd
    ```

    Почему этого не делают?
11. Сколько времени потребуется машине Mic-1, которая работает с частотой 200 МГц, на выполнение следующего высказывания на языке Java: `i=j+k;` Ответ дайте в наносекундах.
12. Тот же вопрос, что и предыдущий, только для машины Mic-2 с частотой 200 МГц. Опираясь на это вычисление, ответьте, сколько времени займет выполнение программы на машине Mic-2, если эта программа выполняется на машине Mic-1 за 100 нс?
13. На машине JVM существуют специальные 1-байтные коды операций для загрузки в стек локальных переменных от 0 до 3, которые используются вместо обычной команды `ILOAD`. Какие изменения нужно внести в машину IJVM, чтобы наилучшим образом использовать эти команды?
14. Команда `ISHR` (целочисленный арифметический сдвиг вправо) есть в машине JVM, но ее нет в машине IJVM. Команда берет два верхних слова стека и заменяет их одним словом (результатом). Второе сверху слово стека — это операнд, который нужно сдвинуть. Он сдвигается вправо на значение от 0 до 31 включительно, в зависимости от значения пяти самых младших битов верхнего слова в стеке (остальные 27 битов игнорируются). Знаковый бит дублируется вправо на столько же битов, на сколько осуществляется сдвиг. Код операции для команды `ISHR` 122 (0x7A).
    1. Какая арифметическая операция эквивалентна сдвигу вправо на 2?
    2. Расширьте систему микрокоманд, чтобы включить эту команду в IJVM.
15. Команда `ISHR` (целочисленный сдвиг влево) имеется в JVM, но отсутствует в IJVM. Команда берет два верхних слова стека и замещает их одним значением (результатом). Второе сверху слово в стеке — операнд, который нужно сдвинуть. Он сдвигается влево на значение от 0 до 31 включительно, в зависимости от значения пяти младших бит верхнего слова в стеке (остальные 2 бита верхнего слова игнорируются). Нули сдвигаются влево на столько же битов, на сколько осуществляется сдвиг. Код операции `ISHL` 120 (0x78).
    1. Какая арифметическая операция эквивалентна сдвигу влево на 2?
    2. Расширьте систему микрокоманд, чтобы включить эту команду в систему IJVM.
16. Команде `INVOKEVIRTUAL` в машине JVM нужно знать, сколько у нее параметров. Зачем?

17. Напишите микропрограмму для Mic-1, чтобы реализовать команду JVM `POPTWO`. Эта команда убирает два верхних слова из стека.
18. Реализуйте команду JVM `DLOAD` для Mic-2. Эта команда содержит 1-байтный индекс и помещает локальную переменную, находящуюся в этом месте, в стек. Затем она помещает следующее старшее слово в стек.
19. Нарисуйте конечный автомат для учета очков при игре в теннис. Правила игры в теннис следующие. Чтобы выиграть, вам нужно получить как минимум 4 очка и у вас должно быть как минимум на 2 очка больше, чем у вашего соперника. Начните с состояния (0, 0), то есть с того, что ни у кого из вас еще нет очков. Затем добавьте состояние (1, 0). Это значит, что игрок *A* получил очко. Дугу из состояния (0, 0) к состоянию (1, 0) обозначьте буквой *A*. Затем добавьте состояние (0, 1), чтобы показать, что игрок *B* получил очко, а дугу к состоянию (0, 1) обозначьте буквой *B*. Продолжайте добавлять состояния и дуги до тех пор, пока не нарисуете все возможные состояния.
20. Вернитесь к предыдущему вопросу. Существуют ли такие состояния, которые могут выйти из строя, но при этом никак не повлияют на результат любой игры? Если да, то какие из них эквивалентны?
21. Нарисуйте конечный автомат для прогнозирования ветвления, более надежный, чем тот, который изображен на рис. 4.29. Он должен изменять предсказание только после трех последовательных неудачных предсказаний.
22. Сдвиговый регистр, изображенный на рис. 4.18, имеет максимальную емкость 6 байтов. Можно ли сконструировать более дешевый блок выборки команд с 5-байтным сдвиговым регистром? А с 4-байтным?
23. Предыдущий вопрос связан с более дешевыми блоками выборки команд. Теперь рассмотрим более дорогие. Встанет ли когда-нибудь вопрос о том, чтобы сконструировать сдвиговый регистр гораздо большей емкости, скажем, 12 байтов? Если да, то почему? Если нет, то почему?
24. В микропрограмме для микроархитектуры Mic-2 микрокоманда `if_icmpeq6` совершает переход к T, если Z установлено на 1. Однако микрокоманда T та же, что и `goto1`. А возможно ли перейти к `goto1` сразу, и станет ли машина работать быстрее после этого?
25. В микроархитектуре Mic-4 блок декодирования отображает код операции IJVM в индекс ПЗУ, где хранятся соответствующие микрооперации. Кажется, что было бы проще опустить стадию декодирования и сразу передать код операции IJVM в очередь. Тогда можно использовать код операции IJVM в качестве индекса в ПЗУ, точно так же, как в микроархитектуре Mic-1. Что не так в этом плане?
26. Компьютер содержит двухуровневую кэш-память. Предположим, что 80% обращений к памяти — удачные обращения в кэш-память первого уровня, 15% — в кэш-память второго уровня, а 5% — промахи кэша. Время доступа составляет 5 нс, 15 нс и 60 нс соответственно, причем время доступа в кэш-память второго уровня и в основную память отсчитывается с того момента, как стало известно, что они нужны (например, доступ к кэш-памяти второго уровня не может начаться, пока не произойдет промах кэш-памяти первого уровня). Каково среднее время доступа?

27. В конце раздела «Кэш-память» мы сказали, что заполнение по записи выгодно только в том случае, если имеют место повторные записи в одну и ту же строку кэш-памяти. А если после записи следуют многочисленные считывания из одной и той же строки, не будет ли заполнение по записи также большим преимуществом?
28. В черновом варианте этой книги на рис. 4.27 вместо 4-входовой ассоциативной кэш-памяти была изображена 3-входовая ассоциативная кэш-память. Один из рецензентов заявил, что читателей это может сильно смутить, поскольку три — это не степень двойки, а компьютеры все делают в двоичной системе. Поскольку потребитель всегда прав, рисунок изменили на 4-входовую ассоциативную кэш-память. Был ли рецензент прав? Аргументируйте.
29. Компьютер с конвейером из пяти стадий при обработке условных переходов простаивает следующие три цикла. Насколько эти простаивания снизят производительность, если 20% команд являются условными переходами? Другие причины простаиваний не учитывайте.
30. Предположим, что компьютер вызывает до 20 команд заранее. В среднем 4 из этих команд являются условными переходами, причем вероятность правильного прогнозирования каждого из этих условных переходов равно 90%. Какова вероятность, что предварительный вызов команд на правильном пути?
31. Предположим, что нам пришлось изменить структуру машины, показанную в табл. 4.12, чтобы использовать 16 регистров вместо 8. Тогда мы изменим команду 6, чтобы использовать регистр R8 в качестве ее выходного регистра. Что в этом случае будет происходить в циклах, начиная с цикла 6?
32. Обычно взаимозависимости затрудняют работу конвейеризированных процессоров. Можно ли что-нибудь сделать с WAW-взаимозависимостью, чтобы улучшить положение вещей? Какие существуют средства оптимизации?
33. Перепишите интерпретатор Mic-1 таким образом, чтобы регистр LV указывал на первую локальную переменную, а не на связующий указатель.
34. Напишите моделирующую программу для одновходовой кэш-памяти прямого отображения. Сделайте число элементов и длину строки параметрами программы. Поэкспериментируйте с этой программой и изложите полученные данные.

Глава 5
Уровень архитектуры команд

В этой главе подробно обсуждается уровень архитектуры команд. Он расположен между микроархитектурным уровнем и уровнем операционной системы, как показано на рис. 1.2. Исторически этот уровень развился прежде всех остальных уровней и изначально был единственным. В наши дни этот уровень очень часто называют «архитектурой» машины, а иногда (что неправильно) «языком ассемблера».

Уровень архитектуры команд имеет особое значение: он является связующим звеном между программным и аппаратным обеспечением. Конечно, можно было бы сделать так, чтобы аппаратное обеспечение сразу непосредственно выполняло программы, написанные на C, C++, FORTRAN 90 или других языках высокого уровня, но это не очень хорошая идея. Преимущество компиляции перед интерпретацией было бы тогда потеряно. Кроме того, из чисто практических соображений компьютеры должны уметь выполнять программы, написанные на разных языках, а не только на одном.

В сущности, все разработчики считают, что нужно транслировать программы, написанные на различных языках высокого уровня, в общую промежуточную форму — на уровень архитектуры команд — и соответственно конструировать аппаратное обеспечение, которое может непосредственно выполнять программы этого уровня (уровня архитектуры команд). Уровень архитектуры команд связывает компиляторы и аппаратное обеспечение. Это язык, который понятен и компиляторам, и аппаратному обеспечению. На рис. 5.1 показана взаимосвязь компиляторов, уровня архитектуры команд и аппаратного обеспечения.

В идеале при создании новой машины разработчики архитектуры команд должны консультироваться и с составителями компиляторов, и с теми, кто конструирует аппаратное обеспечение, чтобы выяснить, какими особенностями должен обладать уровень команд. Если составители компилятора требуют наличия какой-то особенности, которую инженеры не могут реализовать, то такая идея не пройдет. Точно так же, если разработчики аппаратного обеспечения хотят ввести в компьютер какую-либо новую особенность, но составители программного обеспечения не знают, как построить программу, чтобы использовать эту особенность, то такой проект никогда не будет воплощен. После долгих обсуждений и моделирования появится уровень команд, оптимизированный для нужных языков программирования, который и будет реализован.

Но все это в теории. А теперь перейдем к суровой реальности. Когда появляется новая машина, первый вопрос, который задают все потенциальные покупатели: «Совместима ли машина с предыдущими версиями?». Второй вопрос: «Могу ли я

запустить на ней мою старую операционную систему?» И третий вопрос: «Будут ли работать мои прикладные программы на этой машине и не потребуется ли их изменять?» Если какой-нибудь из этих вопросов получает ответ «нет», разработчики должны будут объяснить, почему. Покупатели редко рвутся выбросить все старое программное обеспечение и начать все заново.



Рис. 5.1. Уровень команд — это промежуточное звено между компиляторами и аппаратным обеспечением

Этот факт заставляет компьютерных разработчиков сохранять один и тот же уровень команд в разных моделях или, по крайней мере, делать его **обратно совместимым**. Под обратной совместимостью мы понимаем способность новой машины выполнять старые программы без изменений. Тем не менее новая машина может содержать новые команды и другие особенности, которые могут использоваться новым программным обеспечением. Разработчики должны делать уровень команд совместимым с предыдущими моделями, но они вправе творить все что угодно с аппаратным обеспечением, поскольку едва ли кого-нибудь из покупателей волнует, что собой представляет реальное аппаратное обеспечение и какие действия оно выполняет. Они могут переходить от микропрограммной разработки к непосредственному выполнению, добавлять конвейеры, суперскалярные устройства и т. п., при условии что сохранится обратная совместимость с предыдущим уровнем команд. Основная цель — убедиться, что старые программы работают на новой машине. Тогда возникает проблема: построение лучших машин, но с обратной совместимостью.

Все это вовсе не значит, что разработка уровня команд не имеет никакого значения. Хорошо разработанный уровень архитектуры команд имеет огромные преимущества перед плохим, особенно в отношении вычислительных возможностей и стоимости. Производительность эквивалентных машин с различными уровнями команд может различаться на 25%. Мы просто хотим сказать, что рынок несколько затрудняет (хотя и не делает невозможным) устранение старой архитектуры команд и введение новой. Тем не менее иногда появляются новые уровни команд универсального назначения, а на специализированных рынках (например, на рынке встроенных систем или на рынке мультимедийных процессоров) они возникают гораздо чаще. Следовательно, важно понимать принципы разработки этого уровня.

Какую архитектуру команд можно считать хорошей? Существует два основных фактора. Во-первых, хорошая архитектура должна определять набор команд, которые можно эффективно реализовать в современной и будущей технике, что приводит к рентабельным разработкам на несколько поколений. Плохой проект реализовать сложнее. При плохо разработанной архитектуре команд может потребоваться большее количество вентилей для процессора и больший объем памяти для выполнения программ. Кроме того, машина может работать медленнее, поскольку такая архитектура команд ухудшает возможности перекрывания операций, поэтому для достижения более высокой производительности здесь потребуется более сложный проект. Разработка, в которой используются особенности конкретной техники, может повлечь за собой производство целого поколения компьютеров, и эти компьютеры сможет опередить только более продвинутая архитектура команд.

Во-вторых, хорошая архитектура команд должна обеспечивать ясную цель для оттранслированной программы. Регулярность и полнота вариантов — важные черты, которые не всегда свойственны архитектуре команд. Эти качества важны для компилятора, которому трудно сделать лучший выбор из нескольких возможных, особенно когда некоторые очевидные на первый взгляд варианты не разрешены архитектурой команд. Если говорить кратко, поскольку уровень команд является промежуточным звеном между аппаратным и программным обеспечением, он должен быть удобен и для разработчиков аппаратного обеспечения, и для составителей программного обеспечения.

Общий обзор уровня архитектуры команд

Давайте начнем изучение уровня команд с вопроса о том, что он собой представляет. Этот вопрос на первый взгляд может показаться простым, но на самом деле здесь есть очень много сложностей. В следующем разделе мы обсудим некоторые из этих проблем. Затем мы рассмотрим модели памяти, регистров и команд.

Свойства уровня команд

В принципе уровень команд — это то, каким представляется компьютер программисту машинного языка. Поскольку сейчас ни один нормальный человек не пишет программ на машинном языке, мы переделали это определение. Программа уровня архитектуры команд — это то, что выдает компилятор (в данный момент мы игнорируем вызовы операционной системы и символический язык ассемблера). Чтобы произвести программу уровня команд, составитель компилятора должен знать, какая модель памяти используется в машине, какие регистры, типы данных и команды имеются в наличии и т. д. Вся эта информация в совокупности и определяет уровень архитектуры команд.

В соответствии с этим определением такие вопросы, как программируется ли микроархитектура или нет, конвейеризирован компьютер или нет, является он суперскалярным или нет и т. д., не относятся к уровню архитектуры команд, поскольку составитель компилятора не видит всего этого. Однако это замечание не

совсем справедливо, поскольку некоторые из этих свойств влияют на производительность, а производительность является видимой для программиста. Рассмотрим, например, суперскалярную машину, которая может выдавать back-to-back команды в одном цикле, при условии что одна команда целочисленная, а одна — с плавающей точкой. Если компилятор чередует целочисленные команды и команды с плавающей точкой, то производительность заметно улучшится. Таким образом, детали суперскалярной операции видны на уровне команд, и границы между различными уровнями размыты.

Для одних архитектур уровень команд определяется формальным документом, который обычно выпускается промышленным консорциумом, для других — нет. Например, V9 SPARC (Version 9 SPARC) и JVM имеют официальные определения [156, 85]. Цель такого официального документа — дать возможность различным производителям выпускать машины данного конкретного вида, чтобы эти машины могли выполнять одни и те же программы и получать при этом одни и те же результаты.

В случае с системой SPARC подобные документы нужны для того, чтобы различные предприятия могли выпускать идентичные микросхемы SPARC, отличающиеся друг от друга только производительностью и ценой. Чтобы эта идея работала, поставщики микросхем должны знать, что делает микросхема SPARC (на уровне команд). Следовательно, в документе говорится о том, какая модель памяти, какие регистры присутствуют, какие действия выполняют команды и т. д., а не о том, что представляет собой микроархитектура.

В таких документах содержатся **нормативные** разделы, в которых излагаются требования, и **информативные** разделы, которые предназначены для того, чтобы помочь читателю, но не являются частью формального определения. В нормативных разделах описаны требования и запреты. Например, такое высказывание, как:

выполнение зарезервированного кода операции должно вызывать системное прерывание

означает, что если программа выполняет код операции, который не определен, то он должен вызывать системное прерывание, а не просто игнорироваться. Может быть и альтернативный подход:

результат выполнения зарезервированного кода операции определяется реализацией.

Это значит, что составитель компилятора не может просчитать какие-то конкретные действия, предоставляя конструкторам свободу выбора. К описанию архитектуры часто прилагаются тестовые комплекты для проверки, действительно ли данная реализация соответствует техническим требованиям.

Совершенно ясно, почему V9 SPARC имеет документ, в котором определяется уровень команд: это нужно для того, чтобы все микросхемы V9 SPARC могли выполнять одни и те же программы. По той же причине существует специальный документ для JVM: чтобы интерпретаторы (или такие микросхемы, как picoJava II) могли выполнять любую допустимую программу JVM. Для уровня команд процессора Pentium II такого документа нет, поскольку компания Intel не хочет, чтобы другие производители смогли запускать микросхемы Pentium II. Компания Intel даже обращалась в суд, чтобы запретить производство своих микросхем другими предприятиями.

Другое важное качество уровня команд состоит в том, что в большинстве машин есть, по крайней мере, два режима. **Привилегированный режим** предназначен для запуска операционной системы. Он позволяет выполнять все команды. **Пользовательский режим** предназначен для запуска программных приложений. Этот режим не позволяет выполнять некоторые чувствительные команды (например, те, которые непосредственно манипулируют кэш-памятью). В этой главе мы в первую очередь сосредоточимся на командах и свойствах пользовательского режима.

Модели памяти

Во всех компьютерах память разделена на ячейки, которые имеют последовательные адреса. В настоящее время наиболее распространенный размер ячейки — 8 битов, но раньше использовались ячейки от 1 до 60 битов (см. табл. 2.1). Ячейка из 8 битов называется **байтом**. Причина применения именно 8-битных байтов такова: символы ASCII кода занимают 7 битов, поэтому один символ ASCII плюс бит четности как раз подходит под размер байта. Если в будущем будет доминировать UNICODE, то ячейки памяти, возможно, будут 16-битными. Вообще говоря, число 2^4 лучше, чем 2^3, поскольку 4 — степень двойки, а 3 — нет.

Байты обычно группируются в 4-байтные (32-битные) или 8-байтные (64-битные) слова с командами для манипулирования целыми словами. Многие архитектуры требуют, чтобы слова были выровнены в своих естественных границах. Так, например, 4-байтное слово может начинаться с адреса 0, 4, 8 и т. д., но не с адреса 1 или 2. Точно так же слово из 8 байтов может начинаться с адреса 0, 8 или 16, но не с адреса 4 или 6. Расположение 8-байтных слов показано на рис. 5.2.

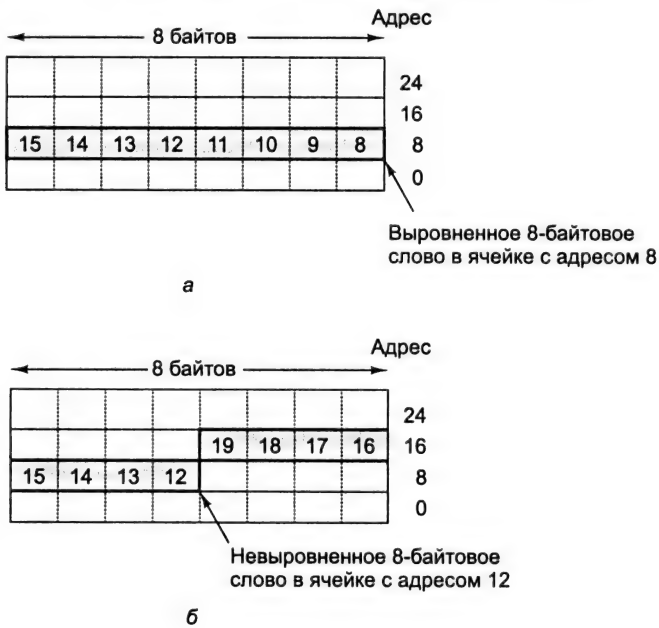


Рис. 5.2. Расположение слова из 8 байтов в памяти: выровненное (*а*); невыровненное (*б*). Некоторые машины требуют, чтобы слова в памяти были выровнены

Выравнивание адресов требуется довольно часто, поскольку при этом память работает более эффективно. Например, Pentium II, который вызывает из памяти по 8 байтов за раз, использует 36-битные физические адреса, но содержит только 33 адресных бита, как показано на рис. 3.41. Следовательно, Pentium II даже не сможет обратиться к невыровненной памяти, поскольку младшие три бита не определены явным образом. Эти биты всегда равны 0, и все адреса памяти кратны 8 байтам.

Тем не менее требование выравнивания адресов иногда вызывает некоторые проблемы. В процессоре Pentium II программы могут обращаться к словам, начиная с любого адреса, — это качество восходит к модели 8088 с шиной данных шириной в 1 байт, в которой не было такого требования, чтобы ячейки располагались в 8-байтных границах. Если программа в процессоре Pentium II считывает 4-байтное слово из адреса 7, аппаратное обеспечение должно сделать одно обращение к памяти, чтобы вызвать байты с 0-го по 7-й, и второе обращение к памяти, чтобы вызвать байты с 8-го по 15-й. Затем центральный процессор должен извлечь требуемые 4 байта из 16 байтов, считанных из памяти, и скомпоновать их в нужном порядке, чтобы сформировать 4-байтное слово.

Возможность считывать слова с произвольными адресами требует усложнения микросхемы, которая после этого становится больше по размеру и дороже. Разработчики были бы рады избавиться от такой микросхемы и просто потребовать, чтобы все программы обращались к словам памяти, а не к байтам. Однако на вопрос инженеров: «Кому нужно исполнение старых программ для машины 8088, которые неправильно обращаются к памяти?» последует ответ продавцов: «Нашим покупателям».

Большинство машин имеют единое линейное адресное пространство, которое простирается от адреса 0 до какого-то максимума, обычно 2^{32} байтов или 2^{64} байтов. В некоторых машинах содержатся отдельные адресные пространства для команд и для данных, так что при вызове команды с адресом 8 и вызове данных с адресом 8 происходит обращение к разным адресным пространствам. Такая система гораздо сложнее, чем единое адресное пространство, но зато она имеет два преимущества. Во-первых, появляется возможность иметь 2^{32} байтов для программы и дополнительные 2^{32} байтов для данных, используя только 32-битные адреса. Во-вторых, поскольку запись всегда автоматически происходит только в пространство данных, случайная перезапись программы становится невозможной, и следовательно, устраняется один из источников программных сбоев.

Отметим, что отдельные адресные пространства для команд и для данных — это не то же самое, что разделенная кэш-память первого уровня. В первом случае все адресное пространство целиком дублируется, и считывание из любого адреса вызывает разные результаты в зависимости от того, что именно считывается: слово или команда. При разделенной кэш-памяти существует только одно адресное пространство, просто в разных блоках кэш-памяти хранятся разные части этого пространства.

Еще один аспект модели памяти — семантика памяти. Естественно ожидать, что команда LOAD, которая встречается после команды STORE и которая обращается к тому же адресу, возвратит только что сохраненное значение. Тем не менее, как мы видели в главе 4, во многих машинах микрокоманды переупорядочиваются.

Таким образом, существует реальная опасность, что память не будет действовать так, как ожидается. Ситуация усложняется в случае с мультипроцессором, когда каждый процессор посылает разделенной памяти поток запросов на чтение и запись, которые тоже могут быть переупорядочены.

Системные разработчики могут применять один из нескольких подходов к этой проблеме. С одной стороны, все запросы памяти могут быть упорядочены в последовательность таким образом, чтобы каждый из них завершался до того, как начнется следующий. Такая стратегия сильно вредит производительности, но зато дает простейшую семантику памяти (все операции выполняются в строгом программном порядке).

С другой стороны, не дается вообще никаких гарантий. Чтобы сделать обращения к памяти упорядоченными, программа должна выполнить команду SYNC, которая блокирует запуск всех новых операций памяти до тех пор, пока предыдущие операции не будут завершены. Эта идея сильно затрудняет работу тех, кто пишет компиляторы, поскольку для этого им нужно очень хорошо знать, как работает соответствующая микроархитектура, но зато разработчикам аппаратного обеспечения предоставлена полная свобода в оптимизации использования памяти.

Возможны также промежуточные модели памяти, в которых аппаратное обеспечение автоматически блокирует запуск определенных операций с памятью (например, тех, которые связаны с RAW- или WAR-взаимозависимостью), при этом запуск всех других операций не блокируется. Хотя разработка этих особенностей на уровне команд довольно утомительна (по крайней мере, для составителей компиляторов и программистов на языке ассемблера), сейчас существует тенденция использовать такой подход. Эта тенденция вызвана такими реализациями, как переупорядочение микрокоманд, конвейеры, многоуровневая кэш-память и т. д. Другие неестественные примеры такого рода мы рассмотрим в этой главе чуть позже.

Регистры

Во всех компьютерах имеется несколько регистров, которые видны на уровне команд. Они нужны там для того, чтобы контролировать выполнение программы, хранить временные результаты, а также для некоторых других целей. Обычно регистры, которые видны на микроархитектурном уровне, например TOS и MAR (см. рис. 4.1), не видны на уровне команд. Тем не менее некоторые из них, например счетчик команд и указатель стека, присутствуют на обоих уровнях. Регистры, которые видны на уровне команд, всегда видны на микроархитектурном уровне, поскольку именно там они реализуются.

Регистры уровня команд можно разделить на две категории: специальные регистры и регистры общего назначения. Специальные регистры включают счетчик команд и указатель стека, а также другие регистры с особой функцией. Регистры общего назначения содержат ключевые локальные переменные и промежуточные результаты вычислений. Их основная функция состоит в том, чтобы обеспечить быстрый доступ к часто используемым данным (обычно избегая обращений к памяти). Машины RISC с высокоскоростными процессорами и медленной (относительно медленной) памятью обычно содержат как минимум 32 регистра общего назначения, а в новых процессорах количество этих регистров постоянно растет.

В некоторых машинах регистры общего назначения полностью симметричны и взаимозаменяемы. Если все регистры эквивалентны, для хранения временного результата компилятор может использовать и регистр R1, и регистр R25. Выбор регистра не имеет никакого значения.

В других машинах некоторые регистры общего назначения могут быть специализированы. Например, в процессоре Pentium II существует регистр EDX, который может использоваться в качестве регистра общего назначения, но который также получает половину произведения и содержит половину делимого при делении.

Даже если регистры общего назначения полностью взаимозаменяемы, операционная система или компиляторы часто принимают соглашения о том, каким образом используются эти регистры. Например, некоторые регистры могут содержать параметры вызываемых процедур, а другие могут использоваться в качестве временных регистров. Если компилятор помещает важную локальную переменную в регистр R1, а затем вызывает библиотечную процедуру, которая воспринимает регистр R1 как временный регистр, доступный для нее, то когда библиотечная процедура возвращает значение, регистр R1 может содержать ненужные данные. А если существуют какие-либо системные соглашения по поводу того, как нужно использовать регистры, составители компиляторов и программисты на языке ассемблера должны следовать им.

Кроме регистров, доступных на уровне команд, всегда существует довольно большое количество специальных регистров, доступных только в привилегированном режиме. Эти регистры контролируют различные блоки кэш-памяти, основную память, устройства ввода-вывода и другие элементы аппаратного обеспечения машины. Данные регистры используются только операционной системой, поэтому компиляторам и пользователям не обязательно знать об их существовании.

Есть один регистр управления, который представляет собой привилегированно-пользовательский гибрид. Это **флаговый регистр**, или **PSW (Program State Word — слово состояния программы**. Этот регистр содержит различные биты, которые нужны центральному процессору. Самые важные биты — это **коды условия**. Они устанавливаются в каждом цикле АЛУ и отражают состояние результата предыдущей операции. Биты кода условия включают:

- N — устанавливается, если результат был отрицательным (Negative);
- Z — устанавливается, если результат был равен 0 (Zero);
- V — устанавливается, если результат вызвал переполнение (oVerflow);
- C — устанавливается, если результат вызвал выход переноса самого левого бита (Carry out);
- A — устанавливается, если произошел выход переноса бита 3 (Auxiliary carry — служебный перенос);
- P — устанавливается, если результат четный (Parity).

Коды условия очень важны, поскольку они используются при сравнениях и условных переходах. Например, команда `CMP` обычно вычитает один операнд из другого и устанавливает коды условия на основе полученной разности. Если

операнды равны, то разность будет равна 0 и во флаговом регистре будет установлен бит Z. Последующая команда BEQ (Branch Equal — переход в случае равенства) проверяет бит Z и совершает переход, если он установлен.

Флаговый регистр содержит не только коды условия. Его содержимое меняется от машины к машине. Дополнительные поля указывают режим машины (например, пользовательский или привилегированный), трассовый бит (который используется для отладки), уровень приоритета процессора, а также статус разрешения прерываний. Флаговый регистр обычно можно считать в пользовательском режиме, но некоторые поля могут записываться только в привилегированном режиме (например, бит, который указывает режим).

Команды

Главная особенность уровня, который мы сейчас рассматриваем, — это набор машинных команд. Они управляют действиями машины. В этом наборе всегда присутствуют команды LOAD и STORE (в той или иной форме) для перемещения данных между памятью и регистрами и команда MOVE для копирования данных из одного регистра в другой. Всегда присутствуют арифметические и логические команды и команды для сравнения элементов данных и переходов в зависимости от результатов. Некоторые типичные команды мы уже рассматривали (см. табл. 4.2.). А в этой главе мы рассмотрим многие другие команды.

Общий обзор уровня команд машины Pentium II

В этой главе мы обсудим три совершенно разные архитектуры команд: IA-32 компании Intel (она реализована в Pentium II), Version 9 SPARC (она реализована в процессорах SPARC) и JVM (она реализована в picoJava II). Мы не преследуем цель дать исчерпывающее описание каждой из этих архитектур. Мы просто хотим продемонстрировать важные аспекты архитектуры команд и показать, как эти аспекты меняются от одной архитектуры к другой. Начнем с машины Pentium II.

Процессор Pentium II развивался на протяжении многих лет. Его история восходит к самым первым микропроцессорам, как мы говорили в главе 1. Основная архитектура команд обеспечивает выполнение программ, написанных для процессоров 8086 и 8088 (которые имеют одну и ту же архитектуру команд), а в машине даже содержатся элементы 8080 — 8-разрядный процессор, который был популярен в 70-е годы. На процессор 8080, в свою очередь, сильно повлияли требования совместимости с процессором 8008, который был основан на процессоре 4004 (4-битной микросхеме, применявшейся еще в каменном веке).

С точки зрения программного обеспечения, компьютеры 8086 и 8088 были 16-разрядными машинами (хотя компьютер 8088 содержал 8-битную шину данных). Их последователь 80286 также был 16-разрядным. Его главным преимуществом был больший объем адресного пространства, хотя небольшое число программ использовали его, поскольку оно состояло из 16 384 64 K сегментов, а не представляло собой линейную 2^{24}-байтную память.

Процессор 80386 был первой 32-разрядной машиной, выпущенной компанией Intel. Все последующие процессоры (80486, Pentium, Pentium Pro, Pentium II, Celeron и Xeon) имеют точно такую же 32-разрядную архитектуру, которая называется **IA-32**, поэтому мы сосредоточим наше внимание именно на этой архитектуре. Единственным существенным изменением архитектуры со времен процессора 80386 было введение команд MMX в более поздние версии системы Pentium и их включение в Pentium II и последующие процессоры.

Pentium II имеет 3 операционных режима, в двух из которых он работает как 8086. В **реальном режиме** все особенности, которые были добавлены к процессору со времен системы 8088, отключаются, и Pentium II работает как простой компьютер 8088. Если программа совершает ошибку, то происходит полный отказ системы. Если бы компания Intel занималась разработкой человеческих существ, то внутрь каждого человека был бы помещен бит, который превращает людей обратно в режим шимпанзе (примитивный мозг, отсутствие речи, питание в основном бананами и т. д.).

На следующей ступени находится **виртуальный режим 8086**, который делает возможным выполнение старых программ, написанных для 8088, с защитой. Чтобы запустить старую программу 8088, операционная система создает специальную изолированную среду, которая работает как процессор 8088, за исключением того, что если программа дает сбой, операционной системе передается соответствующая информация и полного отказа системы не происходит. Когда пользователь WINDOWS начинает работу с MS-DOS, программа, которая действует там, запускается в виртуальном режиме 8086, чтобы программа WINDOWS не могла вмешиваться в программы MS-DOS.

Последний режим — это защищенный режим, в котором Pentium II работает как Pentium II, а не как 8088. В этом режиме доступны 4 уровня привилегий, которые управляются битами во флаговом регистре. Уровень 0 соответствует привилегированному режиму на других компьютерах и имеет полный доступ к машине. Этот уровень используется операционной системой. Уровень 3 предназначен для пользовательских программ. Он блокирует доступ к определенным командам и регистрам управления, чтобы ошибки какой-нибудь пользовательской программы не привели к поломке всей машины. Уровни 1 и 2 используются редко. Pentium II имеет огромное адресное пространство. Память разделена на 16 384 сегмента, каждый из которых идет от адреса 0 до адреса $2^{32}-1$. Однако большинство операционных систем (включая UNIX и все версии WINDOWS) поддерживают только один сегмент, поэтому большинство прикладных программ видят линейное адресное пространство в 2^{32} байтов, а иногда часть этого пространства занимает сама операционная система. Каждый байт в адресном пространстве имеет свой адрес. Слова состоят из 32 битов. Байты нумеруются справа налево (то есть самый первый адрес соответствует самому младшему байту).

Регистры процессора Pentium II показаны на рис. 5.3. Первые четыре регистра EAX, EBX, ECX и EDX 32-битные. Это регистры общего назначения, хотя у каждого из них есть определенные особенности. EAX — основной арифметический регистр; EBX предназначен для хранения указателей (адресов памяти); ECX связан с организацией циклов; EDX нужен для умножения и деления — этот регистр

вместе с EAX содержит 64-битные произведения и делимые. Каждый из этих регистров имеет 16-разрядный регистр в младших 16 битах и 8-разрядный регистр в младших 8 битах. Данные регистры позволяют легко манипулировать 16-битными и 8-битными значениями соответственно. В компьютерах 8088 и 80286 есть только 8-битные и 16-битные регистры. 32-битные регистры появились в системе 80386 вместе с приставкой E (Extended — расширенный).

Следующие три регистра также являются регистрами общего назначения, но с большей степенью специализации. Регистры ESI и EDI предназначены для хранения указателей, особенно для команд манипулирования цепочками, где ESI указывает на входную цепочку, а EDI — на выходную цепочку. Регистр EBP тоже предназначен для хранения указателей. Обычно он используется для указания на основу текущего фрейма локальных переменных, как и регистр LV в машине IJVM. Такой регистр обычно называют **указателем фрейма**. Наконец, регистр ESP — это указатель стека.

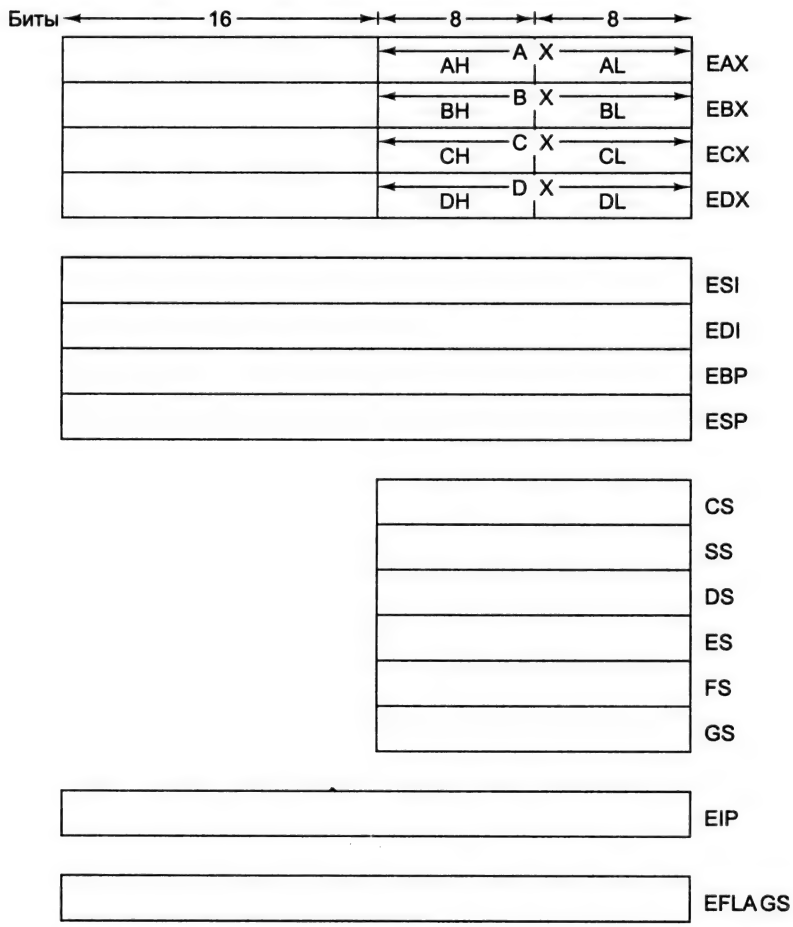


Рис. 5.3. Основные регистры процессора Pentium II

Следующая группа регистров от CS до GS — сегментные регистры. Это электронные трилобиты — атавизмы, оставшиеся от процессора 8088, который обращался к 2^{20} байтам памяти, используя 16-битные адреса. Достаточно сказать, что когда Pentium II установлен на использование единого линейного 32-битного адресного пространства, их можно смело проигнорировать. Регистр EIP — это счетчик программ (Extended Instruction Pointer — расширенный указатель команд). Регистр EFLAGS — это флаговый регистр.

Общий обзор уровня команд системы UltraSPARC II

Архитектура SPARC была впервые введена в 1987 году компанией Sun Microsystems. Эта архитектура была одной из первых архитектур промышленного назначения типа RISC. Она была основана на исследовании, проведенном в Беркли в 80-е годы [110, 113]. Изначально система SPARC была 32-разрядной архитектурой, но UltraSPARC II — это 64-разрядная машина, основанная на Version 9, и именно ее мы будем описывать в этой главе. В целях согласованности с остальными частями книги мы будем называть данную систему UltraSPARC II, хотя на уровне команд все машины UltraSPARC идентичны.

Структура памяти машины UltraSPARC II очень проста: память представляет собой линейный массив из 2^{64} байтов. К сожалению, память настолько велика (18 446 744 073 709 551 616 байтов), что в настоящее время ее невозможно реализовать. Современные реализации имеют ограничение на размер адресного пространства, к которому они могут обращаться (2^{44} байтов у UltraSPARC II), но в будущем это число увеличится. Байты нумеруются слева направо, но нумерацию можно изменить и сделать ее справа налево, установив бит во флаговом регистре.

Важно, что архитектура команд имеет больше байтов, чем требуется для реализации, поскольку в будущем, скорее всего, понадобится увеличить размер памяти, к которой может обращаться процессор. Одна из самых серьезных проблем при разработке архитектур состоит в том, что архитектура команд ограничивает размер адресуемой памяти. В информатике существует один вопрос, который совершенно невозможно разрешить: никогда не хватает того количества битов, которое имеется в данный момент. Когда-нибудь ваши внуки спросят у вас, как же могли работать компьютеры, которые содержат всего-навсего 32-битные адреса и только 4 Гбайт памяти.

Архитектура команд SPARC достаточно проста, хотя организация регистров была немного усложнена, чтобы сделать вызовы процедур более эффективными. Практика показывает, что организация регистров требует больших усилий и в общем эти усилия не стоят того, но правило совместимости не позволяет избавиться от этого.

В системе UltraSPARC II имеется две группы регистров: 32 64-битных регистра общего назначения и 32 регистра с плавающей точкой. Регистры общего назначения называются R0–R31, но в определенных контекстах используются другие названия. Варианты названий регистров и их функции приведены в табл. 5.1.

Таблица 5.1. Регистры общего назначения в системе UltraSPARC II

Регистр	Другой вариант названия	Функция
R0	G0	Связан с 0. То, что сохранено в этом регистре, просто игнорируется
R1–R7	G1–G7	Содержит глобальные переменные
R8–R13	O0–O5	Содержит параметры вызываемой процедуры
R14	SP	Указатель стека
R15	O7	Временный регистр
R16–R23	L0–L7	Содержит локальные переменные для текущей процедуры
R24–R29	I0–I5	Содержит входные параметры
R30	FP	Указатель на основу текущего стекового фрейма
R31	I7	Содержит адрес возврата для текущей процедуры

Все регистры общего назначения 64-битные. Все они, кроме R0, значение которого всегда равно 0, могут считываться и записываться при помощи различных команд загрузки и сохранения. Функции, приведенные в табл. 5.1, отчасти определены по соглашению, но отчасти основаны на том, как аппаратное обеспечение обрабатывает их. Вообще не стоит отклоняться от этих функций, если вы не являетесь крупным специалистом, блестяще разбирающимся в компьютерах SPARC. Программист должен быть уверен, что программа правильно обращается к регистрам и выполняет над ними допустимые арифметические действия. Например, очень легко загрузить числа с плавающей точкой в регистры общего назначения, а затем произвести над ними целочисленное сложение, операцию, которая приведет к полнейшей чепухе, но которую центральный процессор обязательно выполнит, если этого потребует программа.

Глобальные переменные используются для хранения констант, переменных и указателей, которые нужны во всех процедурах, хотя они могут загружаться и перезагружаться при входе в процедуру и при выходе из процедуры, если нужно. Регистры Ix и Ox используются для передачи параметров процедурам, чтобы избежать обращений к памяти. Ниже мы расскажем, как это происходит.

Специальные регистры используются для особых целей. Регистры FP и SP ограничивают текущий фрейм. Первый указывает на основу текущего фрейма и используется для обращения к локальным переменным, точно так же как LV на рис. 4.9. Второй указывает на текущую вершину стека и изменяется, когда слова помещаются в стек или выталкиваются оттуда. Значение регистра FP изменяется только при вызове и завершении процедуры. Третий специальный регистр — R31. Он содержит адрес возврата для текущей процедуры.

В действительности процессор UltraSPARC II имеет более 32 регистров общего назначения, но только 32 из них видны для программы в любой момент времени. Эта особенность, называемая **регистровыми окнами**, предназначена для повышения эффективности вызова процедур. Система регистровых окон проиллюстрирована рис. 5.4. Основная идея — имитировать стек, используя при этом регистры. То есть существует несколько наборов регистров, точно так же как и несколько

фреймов в стеке. Ровно 32 регистра общего назначения видны в любой момент. Регистр CWP (Current Window Pointer — указатель текущего окна) следит за тем, какой набор регистров используется в данный момент.

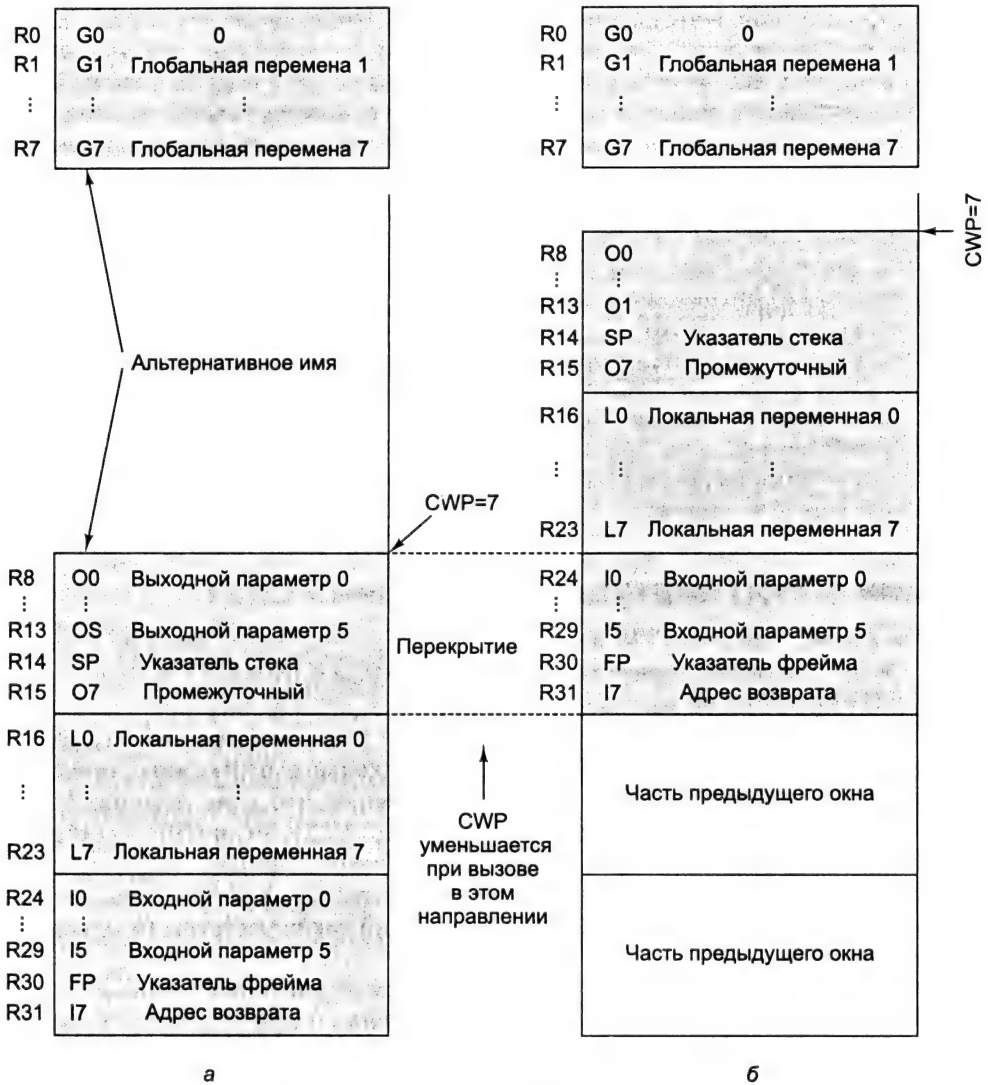


Рис. 5.4. Регистровые окна системы UltraSPARC II

Команда вызова процедуры скрывает старый набор регистров и путем изменения CWP предоставляет новый набор, который может использовать вызванная процедура. Однако некоторые регистры переносятся из вызывающей процедуры к вызванной процедуре, что обеспечивает эффективный способ передачи параметров между процедурами. Для этого некоторые регистры переименовываются: после вызова процедуры прежние выходные регистры с R8 по R15 все еще видны, но

теперь это входные регистры с R24 по R31. Восемь глобальных регистров не меняются. Это всегда один и тот же набор регистров.

В отличие от памяти, которая квазибесконечна (по крайней мере, в отношении стеков), если происходит многократное вложение процедур, машина выходит из регистровых окон. В этот момент самый старый набор регистров сбрасывается в память, чтобы освободить новый набор. Точно так же после многократных выходов из процедур может понадобиться вызвать набор регистров из памяти. В целом такая усложненность является большой помехой и, вообще говоря, не очень полезна. Такая система выгодна только в том случае, если нет многократных вложений процедур.

В системе UltraSPARC II есть также 32 регистра с плавающей точкой, которые могут содержать либо 32-битные (одинарная точность), либо 64-битные (двойная точность) значения. Возможно также использовать пары этих регистров, чтобы поддерживать 128-битные значения.

Архитектура UltraSPARC II — **архитектура загрузки/хранения**. Это значит, что единственные операции, которые непосредственно обращаются к памяти — это команды LOAD (загрузка) и STORE (сохранение), служащие для перемещения данных между регистрами и памятью. Все операнды для команд арифметических и логических действий должны браться из регистров или предоставляться самой командой (но не должны браться из памяти), и все результаты должны сохраняться в регистрах (но не в памяти).

Общий обзор виртуальной машины Java

Уровень команд машины JVM необычен, но достаточно прост. Мы уже отчасти рассмотрели его во время изучения машины JVM. Модель памяти JVM точно такая же, как у IJVM, о которой мы говорили в главе 4 (см. рис. 4.9), но с одной дополнительной областью, о которой мы сейчас расскажем. Порядок байтов обратный.

Память содержит 4 основные области: фрейм локальных переменных, стек операндов, область процедур и набор констант. Напомним, что в реализациях Mic-*x* машины IJVM на эти области указывают регистры LV, SP, PC и CPP. Доступы к памяти должны осуществляться только по смещению от одного из этих регистров; указатели и абсолютные адреса памяти не используются. Хотя JVM не требует наличия этих регистров, в большинстве реализаций такие регистры (или подобные им) имеются.

Отсутствие указателей для доступа к локальным переменным и константам не случайно. Это нужно для достижения одной из главных целей языка Java: возможности загружать двоичную программу из Интернета и выполнять ее, не опасаясь шпионских программ или какого-либо сбоя в машине, на которой это программа выполняется. Если ограничить использование указателей, можно добиться высокой безопасности.

Напомним, что ни одна из областей памяти, определенных в машине IJVM, не может быть очень большой. Объем области процедур может быть всего лишь 64 Кбайт. Пространство, занимаемое локальными переменными, не должно превышать 64 Кбайт. Набор констант также ограничивается 64 Кбайт. JVM характеризуется теми же ограничениями по той же причине: смещения для индексирования этих областей ограничиваются 16 битами.

Область локальных переменных меняется с каждой процедурой, поэтому каждая вызываемая процедура имеет собственные 64 Кбайт для своих локальных переменных. Точно так же определенный набор констант распространяется только на определенный класс, поэтому каждый из них имеет собственные 64 Кбайт. Здесь нет места для хранения больших массивов и динамических структур данных, например списков и деревьев. Именно поэтому в JVM включается дополнительная область памяти, так называемая **«куча»**, которая предназначена для хранения динамических объектов, а также очень крупных объектов. Когда компилятор Java воспринимает следующее выражение:

```
int a[]=new int[4096]
```

он посылает сигнал распределителю памяти, который определяет место в памяти для «кучи» и возвращает ей указатель. Таким образом, указатели используются в JVM, но программисты не могут непосредственно манипулировать ими.

Если в «куче» создаются все новые и новые структуры данных, она в конце концов переполнится. Когда специальная система определяет, что «куча» почти заполнилась, она вызывает **программу чистки памяти (сборщик мусора)**, которая ищет ненужные объекты в «куче». Для этого применяется сложный алгоритм. Ненужные объекты отбрасываются, чтобы освободить место в «куче». Схема действия автоматической программы чистки памяти полностью отличается от использования стека локальных переменных. Эти два метода находятся в отношении дополнительности; каждый из них имеет свое место.

JVM не содержит регистров общего назначения, которые могут загружаться или сохраняться под управлением программы. Это машина со стековой организацией. Хотя для наших дней это необычно, такая машина дает простую архитектуру команд, которую легко компилировать.

Однако у машины со стековой организацией есть один недостаток: здесь требуется большое количество обращений к памяти. Но, как мы видели в разделе «Микроархитектура процессора picoJava II» главы 4, при умелой разработке можно устранить большую часть из них путем свертывания команд JVM. Более того, поскольку данная архитектура очень проста, ее можно реализовать в небольшом кусочке кремния, оставив большую часть пространства микросхемы свободной для кэш-памяти первого уровня, которая в дальнейшем сократит число обращений к основной памяти. (Размер кэш-памяти picoJava II составляет максимум 16 Кбайт + 16 Кбайт, поскольку основной целью было создание дешевой микросхемы.)

Типы данных

Всем компьютерам нужны данные. Для многих компьютерных систем основной задачей является обработка финансовых, промышленных, научных, технических и других данных. Внутри компьютера данные должны быть представлены в какой-либо особой форме. На уровне архитектуры команд используются различные типы данных. Они будут описаны ниже.

Ключевым вопросом является вопрос о том, существует ли аппаратная поддержка для конкретного типа данных. Под аппаратной поддержкой подразумевается, что одна или несколько команд ожидают данные в определенном формате и пользо-

ватель не может брать другой формат. Например, бухгалтеры привыкли писать знак «минус» справа у отрицательных чисел, а специалисты по вычислительной технике — слева. Предположим, что, пытаясь произвести впечатление на своего начальника, глава компьютерного центра в бухгалтерской фирме изменил все числа во всех компьютерах, чтобы знаковый бит был самым правым битом (а не самым левым). Несомненно, это произведет большое впечатление на начальника, поскольку все программное обеспечение больше не будет функционировать правильно. Аппаратное обеспечение требует определенного формата для целых чисел и не будет работать должным образом, если целые числа поступают в другом формате.

Теперь рассмотрим другую бухгалтерскую фирму, которая только что заключила договор на проверку федерального долга. 32-битная арифметика здесь не подойдет, поскольку числа превышают 2^{32} (около 4 миллиардов). Одно из возможных решений — использовать два 32-битных целых числа для представления каждого числа, то есть все 64 бита. Если машина не поддерживает такие **числа с удвоенной точностью**, то все арифметические операции над ними должны выполняться программным обеспечением, но эти две части могут располагаться в произвольном порядке, поскольку для аппаратного обеспечения это не важно. Это пример типа данных без аппаратной поддержки и, следовательно, без аппаратной реализации. В следующих разделах мы рассмотрим типы данных, которые поддерживаются аппаратным обеспечением и для которых требуются специальные форматы.

Числовые типы данных

Типы данных можно разделить на две категории: числовые и нечисловые. Среди числовых типов данных главными являются целые числа. Они бывают различной длины: обычно 8, 16, 32 и 64 бита. Целые числа применяются для подсчета различных предметов (например, они показывают, сколько на складе имеется отверток), для идентификации различных объектов (например, банковских счетов), а также для других целей. В большинстве современных компьютеров целые числа хранятся в двоичной записи, хотя в прошлом использовались и другие системы. Двоичные числа обсуждаются в приложении А.

- Некоторые компьютеры поддерживают целые числа и со знаком, и без знака. В целом числе без знака нет знакового бита, и все биты содержат данные. Этот тип данных имеет преимущество: у него есть дополнительный бит, поэтому 32-битное слово может содержать целое число без знака от 0 до $2^{32}-1$ включительно. Двоичное целое число со знаком, напротив, может содержать числа только до $2^{31}-1$, но зато включает и отрицательные числа.

Для выражения нецелых чисел (например, 3,5) используются числа с плавающей точкой. Они обсуждаются в приложении Б. Их длина составляет 32, 64, а иногда и 128 битов. В большинстве компьютеров есть команды для выполнения операций с числами с плавающей точкой. Во многих компьютерах имеются отдельные регистры для целочисленных операндов и для операндов с плавающей точкой.

Некоторые языки программирования, в частности COBOL, допускают в качестве типа данных десятичные числа. Машины, предназначенные для программ на языке COBOL, часто поддерживают десятичные числа в аппаратном обеспечении,

обычно кодируя десятичный разряд в 4 бита и затем объединяя два десятичных разряда в байт (двоично-десятичный формат). Однако результаты арифметических действий над такими десятичными числами будут некорректны, поэтому требуются специальные команды для коррекции десятичной арифметики. Эти команды должны знать выход переноса бита 3. Вот почему код условия часто содержит бит служебного переноса. Между прочим, проблема 2000 года была вызвана программистами на языке COBOL, которые решили, что дешевле будет представлять год в виде двух десятичных разрядов, а не в виде 16-битного двоичного числа.

Нечисловые типы данных

Хотя самые первые компьютеры работали в основном с числами, современные компьютеры часто используются для нечисловых приложений, например, для обработки текстов или управления базой данных. Для этих приложений нужны другие, нечисловые, типы данных. Они часто поддерживаются командами уровня архитектуры команд. Здесь очень важны символы, хотя не каждый компьютер обеспечивает аппаратную поддержку для них. Наиболее распространенными символьными кодами являются ASCII и UNICODE. Они поддерживают 7-битные и 16-битные символы соответственно. Эти коды обсуждались в главе 2.

На уровне команд часто имеются особые команды, предназначенные для операций с цепочками символов. Эти цепочки иногда разграничиваются специальным символом в конце. Вместо этого для определения конца цепочки может использоваться поле длины цепочки. Команды могут выполнять копирование, поиск, редактирование цепочек и другие действия.

Кроме того, важны значения булевой алгебры. Этих значений два: истина и ложь. Теоретически булево значение может представлять один бит: 0 — ложь, а 1 — истина (или наоборот). На практике же используется байт или слово, поскольку отдельные биты в байте не имеют собственных адресов, и следовательно, к ним трудно обращаться. В обычных системах применяется следующее соглашение: 0 означает ложь, а все остальное означает истину.

Единственная ситуация, в которой булево значение представлено 1 битом, — это когда имеется целый массив бит и 32-битное слово может содержать 32 булевых значения. Такая структура данных называется **битовым отображением**. Она встречается в различных контекстах. Например, битовое отображение может использоваться для того, чтобы следить за свободными блоками на диске. Если диск содержит n блоков, тогда битовое отображение содержит n битов.

Последний тип данных — это указатели, которые представляют собой машинные адреса. Мы уже неоднократно рассматривали указатели. В машинах Mic-*x* регистры SP, PC, LV и CPP — это примеры указателей. Доступ к переменной на фиксированном расстоянии от указателя (а именно так работает команда `ILOAD`) широко применяется на всех машинах.

Типы данных процессора Pentium II

Pentium II поддерживает двоичные целые числа со знаком, целые числа без знака, числа двоично-десятичной системы счисления и числа с плавающей точкой по стандарту IEEE 754 (табл. 5.2). Эта машина является 8-, 16-разрядной и оперирует с

целыми числами такой длины. У нее имеются многочисленные арифметические команды, булевы операции и операции сравнения. Операнды необязательно должны быть выровнены в памяти, но если адреса слов кратны 4 байтам, то наблюдается более высокая производительность.

Таблица 5.2. Числовые типы данных процессора Pentium II. Поддерживаемые типы отмечены крестом (×)

Тип	8 битов	16 битов	32 бита	64 бита	128 битов
Целые числа со знаком	×	×	×		
Целые числа без знака	×	×	×		
Двоично-десятичные целые числа	×				
Числа с плавающей точкой			×	×	

Pentium II также может манипулировать 8-разрядными символами ASCII: существуют специальные команды для копирования и поиска цепочек символов. Эти команды используются и для цепочек, длина которых известна заранее, и для цепочек, в конце которых стоит специальный маркер. Они часто используются в библиотеках операций над строками.

Типы данных машины UltraSPARC II

UltraSPARC II поддерживает широкий ряд форматов данных (табл. 5.3). Эта машина может поддерживать 8-, 16-, 32- и 64-битные целочисленные операнды со знаком и без знака. Целые числа со знаком используют дополнительный код. Кроме того, имеются операнды с плавающей точкой по 32, 64 и 128 битов, которые соответствуют стандарту IEEE 754 (для 32-битных и 64-битных чисел). Двоично-десятичные числа не поддерживаются. Все операнды должны быть выровнены в памяти.

Таблица 5.3. Числовые типы данных компьютера UltraSPARC II

Тип	8 битов	16 битов	32 бита	64 бита	128 битов
Целые числа со знаком	×	×	×	×	
Целые числа без знака	×	×	×	×	
Двоично-десятичные целые числа					
Числа с плавающей точкой			×	×	×

UltraSPARC II представляет собой регистровую структуру, и почти все команды оперируют с 64-разрядными регистрами. Символьные и строковые типы данных специальными командами аппаратного обеспечения не поддерживаются.

Типы данных виртуальной машины Java

Java — это язык со строгим контролем типов. Это значит, что каждый операнд имеет особый тип и размер, который известен в период компиляции. Это отражено в типах, поддерживаемых JVM. JVM поддерживает числовые типы, приведенные в табл. 5.4. Целые числа со знаком используют дополнительный код.

Целые числа без знака в языке Java не присутствуют и не поддерживаются JVM, как и двоично-десятичные числа.

Таблица 5.4. Числовые типы данных для JVM

Тип	8 битов	16 битов	32 бита	64 бита	128 битов
Целые числа со знаком	×	×	×	×	
Целые числа без знака					
Двоично-десятичные целые числа					
Числа с плавающей точкой			×	×	

JVM поддерживает символы, но не традиционные 8-битные символы ASCII, а 16-битные символы UNICODE. Указатели поддерживаются главным образом для внутреннего использования компилятора и системы обслуживания. Пользовательские программы не могут непосредственно обращаться к указателям. Указатели используются в основном для ссылок на объекты.

Форматы команд

Команда состоит из кода операции и некоторой дополнительной информации, например, откуда поступают операнды и куда должны отправляться результаты. Процесс определения, где находятся операнды (то есть их адреса), называется **адресацией.**

На рисунке 5.5 показано несколько возможных форматов для команд второго уровня. Команды всегда содержат код операции, который сообщает, какие действия выполняет команда. В команде может присутствовать ноль, один, два или три адреса.



Рис. 5.5. Четыре формата команд: безадресная команда (*а*); одноадресная команда (*б*); двухадресная команда (*в*); трехадресная команда (*г*)

В одних машинах все команды имеют одинаковую длину; в других команды могут быть разной длины. Команды могут быть короче слова, длиннее слова или быть равными слову по длине. Если все команды одной длины, то это упрощает декодирование, но часто требует большего пространства, поскольку все команды должны быть такой же длины, как самая длинная. На рис. 5.6 показано несколько возможных соотношений между длиной команды и длиной слова.

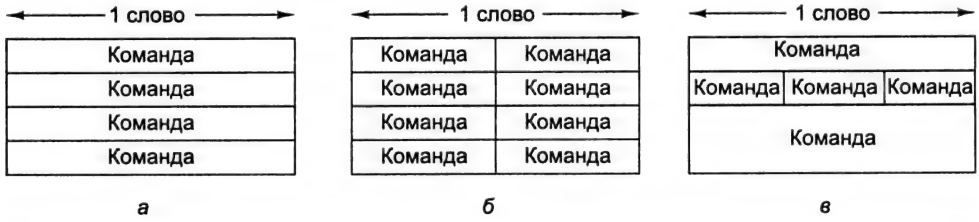


Рис. 5.6. Некоторые возможные отношения между длиной команды и длиной слова

Критерии разработки для форматов команд

Если разработчикам нужно выбрать форматы команд для их машины, они должны рассмотреть ряд факторов. Нельзя недооценивать сложность этого решения. Если компьютер получился удачным с коммерческой точки зрения, то набор команд может продолжать существовать 20 лет и более. Имеет огромное значение способность добавлять новые команды и использовать другие возможности, которые появляются на протяжении определенного периода времени, но только в том случае, если архитектура и компания, создавшая эту архитектуру, существуют достаточно долго.

Эффективность конкретной архитектуры команд зависит от технологии, которая применялась при разработке компьютера. За длительный период времени эта технология будет подвергнута значительным изменениям, и некоторые характеристики архитектуры команд окажутся неудачными (если оглянуться на прошлое). Например, если доступ к памяти осуществляется быстро, то подойдет стековая архитектура (как в JVM), но если доступ к памяти медленный, тогда желательно иметь много регистров (как в UltraSPARC II). Тем читателям, которые считают, что этот выбор сделать просто, мы предлагаем взять лист бумаги и записать следующие предположения: 1) какова будет типичная скорость тактового генератора через 20 лет и 2) каково будет типичное время доступа к ОЗУ через 20 лет. Аккуратно сложите этот лист бумаги и спрячьте его в надежном месте, а через 20 лет разверните и прочитайте, что на нем написано. Те из вас, кто принял этот вызов, могут не использовать лист бумаги, а просто отправить свои предсказания в Интернет.

Даже дальновидные разработчики не всегда могут сделать правильный выбор. И даже если бы они могли его сделать, они бы просуществовали недолго, поскольку если такая развитая архитектура команд будет стоить дороже, чем архитектуры у конкурентов, то компания долго не продержится.

Если речь идет об одинаковых машинах, то лучше иметь короткие команды, чем длинные. Программа, состоящая из n 16-битных команд, занимает в два раза меньше пространства памяти, чем программа из n 32-битных команд. Поскольку цены на память постоянно падают, этот фактор не имел бы значения в будущем, если бы программное обеспечение не разрасталось гораздо быстрее, чем происходит снижение цен на память.

Более того, минимизация размера команд может усложнить их декодирование и перекрытие. Следовательно, достижение минимального размера команды должно уравниваться со временем, затрачиваемым на декодирование и выполнение команд.

Есть еще одна очень важная причина минимизации длины команд, и она становится все важнее с увеличением скорости работы процессоров: пропускная способность памяти (число битов в секунду, которое память может предоставлять). Значительный рост скорости работы процессора за последнее десятилетие не соответствует увеличению пропускной способности памяти. Ограничения здесь связаны с неспособностью системы памяти передавать команды и операнды с той же скоростью, с какой процессор может обрабатывать их. Пропускная способность памяти зависит от технологии разработки. Трудности, связанные с пропускной способностью, имеют отношение не только к основной памяти, но и ко всем видам кэш-памяти.

Если пропускная способность кэш-памяти команд составляет t бит/с, а средняя длина команды r битов, то кэш-память способна передавать самое большее t/r команд в секунду. Отметим, что это *верхний предел* скорости, с которой процессор может выполнять команды, хотя в настоящее время предпринимаются попытки преодолеть этот барьер. Ясно, что скорость, с которой могут выполняться команды (то есть скорость работы процессора), может ограничиваться длиной команд. Чем короче команды, тем быстрее работает процессор. Поскольку современные процессоры способны выполнять несколько команд за один цикл, то вызов нескольких команд за цикл обязателен. Этот аспект кэш-памяти команд делает размер команд важным критерием, который нужно учитывать при разработке.

Второй критерий разработки — достаточный объем пространства в формате команды для выражения всех требуемых операндов. Машина с 2^n операциями и со всеми командами менее n битов невозможна. В этом случае в коде операции не было бы достаточно места для того, чтобы указать, какая нужна команда. И история снова и снова доказывает, что обязательно нужно оставлять большое количество свободных кодов операций для будущих дополнений к набору команд.

Третий критерий связан с числом битов в адресном поле. Рассмотрим проект машины с 8-битными символами и основной памятью, которая должна содержать 2^{32} символов. Разработчики вольны были приписать последовательные адреса блокам по 8, 16, 24 или 32 бита.

Представим, что бы случилось, если бы команда разработчиков разбилась на две воюющие группы, одна из которых утверждает, что основной единицей памяти должен быть 8-битный байт, а другая требует, чтобы основной единицей памяти было 32-битное слово. Первая группа предложила бы память из 2^{32} байтов с номерами 0, 1, 2, 3, ..., 4 294 967 295. Вторая группа предложила бы память из 2^{30} слов с номерами 0, 1, 2, 3, ..., 1 073 741 823.

Первая группа скажет, что для того чтобы сравнить два символа при организации по 32-битным словам, программе приходится не только вызывать из памяти слова, содержащие эти символы, но и выделять соответствующий символ из каждого слова для сравнения. А это потребует наличия дополнительных команд и, следовательно, дополнительного пространства. 8-битная организация, напротив, обеспечивает адрес для каждого символа, что значительно упрощает процедуру сравнения.

Сторонники 32-битной организации скажут, что их проект требует всего лишь 2^{30} отдельных адресов, что дает длину адреса всего 30 битов, тогда как при 8-битной организации требуется целых 32 бита для обращения к той же самой памяти.

Если адрес короткий, то и команда будет более короткой. Она будет занимать меньше пространства в памяти, и к тому же для ее вызова потребуется меньше времени. В качестве альтернативы они могут сохранить 32-битный адрес для обращения к памяти в 16 Гбайт вместо каких-то там 4 Гбайт.

Этот пример демонстрирует, что для получения оптимальной дискретности памяти требуются более длинные адреса и, следовательно, более длинные команды. Одна крайность — это организация памяти, при которой адресуется каждый бит (например, Burroughs B1700). Другая крайность — это память, состоящая из очень длинных слов (например, серия CDC Cyber содержала 60-битные слова).

Современные компьютерные системы пришли к компромиссу, который, в каком-то смысле, объединил в себе худшие качества обоих вариантов. Они требуют, чтобы адреса были у отдельных байтов, но при обращении к памяти всегда считывается одно, два, а иногда даже четыре слова сразу. В результате считывания одного байта из памяти на машине UltraSPARC II вызывается сразу минимум 16 байтов (см. рис. 3.44), а иногда и вся строка кэш-памяти в 64 байта.

Расширение кода операций

В предыдущем разделе мы увидели, что короткие адреса противостоят удачной дискретности памяти. В этом разделе мы рассмотрим компромиссы, связанные с кодами операций и адресами. Рассмотрим команду размером (n+k) битов с кодом операции в k битов и одним адресом в n битов. Такая команда допускает 2^k различных операций и 2^n адресуемых ячеек памяти. В качестве альтернативы те же (n+k) битов можно разбить на код операции в (k−1) битов и адрес в (n+1) битов. При этом будет либо в два раза меньше команд, но в два раза больше памяти, либо то же количество памяти, но дискретность вдвое выше. Код операции на (k+1) битов и адрес в (n−1) битов дает большее количество операций, но ценой этого преимущества является либо меньшее количество ячеек памяти, либо не очень удачная дискретность при том же объеме памяти. Наряду с простыми компромиссами между битами кода операции и битами адреса, которые были только что описаны, возможны и более сложные. Схема, которая будет обсуждаться в следующих разделах, называется **расширением кода операций**.

Понятие расширения кода операций можно пояснить на примере. Рассмотрим машину, в которой длина команд составляет 16 битов, а длина адресов — 4 бита, как показано на рис. 5.7. Эта ситуация вполне разумна для машины, содержащей 16 регистров (а следовательно, 4-битный адрес регистра), над которыми совершаются все арифметические операции. Один из возможных вариантов — наличие в каждой команде 4-битного кода операции и трех адресов, что дает 16 трехадресных команд.

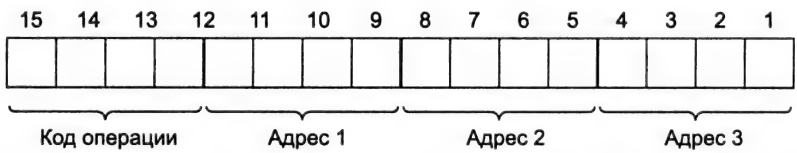


Рис. 5.7. Команда с 4-битным кодом операции и тремя 4-битными адресными полями

Если разработчикам нужно 15 трехадресных команд, 14 двухадресных команд, 31 одноадресная команда и 16 безадресных команд, они могут использовать коды операций от 0 до 14 в качестве трехадресных команд, а код операции 15 уже интерпретировать по-другому (рис. 5.8).

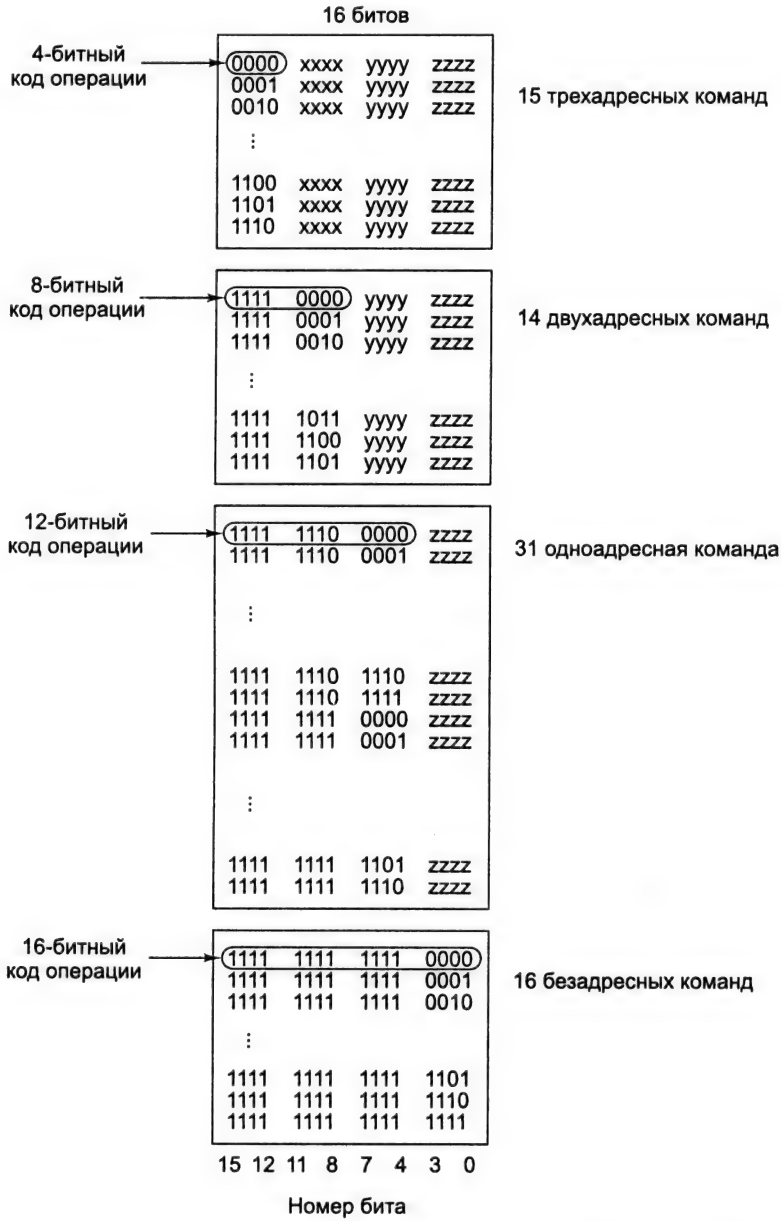


Рис. 5.8. Расширение кода операции допускает 15 трехадресных команд, 14 двухадресных команд, 31 одноадресную команду и 16 безадресных команд. Поля *xxxx*, *yyyy* и *zzzz* — это 4-битные адресные поля

Это значит, что код операции 15 содержится в битах с 8 по 15, а не с 12 по15. Биты с 0 по 3 и с 4 по 7, как и раньше, формируют два адреса. Все 14 двухадресных команд содержат число 1111 в старших четырех битах и числа от 0000 до 1101 в битах с 8 по 11. Команды с числом 1111 в старших четырех битах и числом 1110 или 1111 в битах с 8 по 11 будут рассматриваться особо. Они будут трактоваться таким образом, как будто их коды операций находятся в битах с 4 по 15. В результате получаем 32 новых кода операций. А поскольку требуется всего 31 код, то код 11111111111 означает, что действительный код операции находится в битах с 0 по 15, что дает 16 безадресных команд.

Как видим, код операции становится все длиннее и длиннее: трехадресные команды имеют 4-битный код операции, двухадресные команды — 8-битный код операции, одноадресные команды — 12-битный код операции, а безадресные команды — 16-битный код операции.

Идея расширения кода операций наглядно демонстрирует компромисс между пространством для кодов операций и пространством для другой информации. Однако на практике все не так просто и понятно, как в нашем примере. Есть только два способа изменения размера кодов операций. С одной стороны, можно иметь все команды одинаковой длины, приписывая самые короткие коды операций тем командам, которым нужно больше всего битов для спецификации чего-либо другого. С другой стороны, можно свести к минимуму *средний* размер команды, если выбрать самые короткие коды операций для часто используемых команд и самые длинные — для редко используемых команд.

Если довести эту идею до конца, можно свести к минимуму среднюю длину команды, закодировав каждую команду, чтобы максимально уменьшить число требуемых битов. К сожалению, это приведет к наличию команд разных размеров, которые не будут выровнены в границах байтов. Пока существуют архитектуры команд с таким свойством (например Intel 432), выравнивание будет иметь большое значение для быстрого декодирования команд. Тем не менее эта идея часто применяется на уровне байтов. Ниже мы рассмотрим архитектуру команд JVM, чтобы показать, как можно менять форматы команд, чтобы максимально уменьшить размер программы.

Форматы команд процессора Pentium II

Форматы команд процессора Pentium II очень сложны и нерегулярны. Они содержат до шести полей разной длины, пять из которых факультативны. Общая модель показана на рис. 5.9. Эта ситуация сложилась из-за того, что архитектура развивалась на протяжении нескольких поколений и ранее в нее были включены не очень удачно выбранные характеристики. Из-за требования обратной совместимости позднее их нельзя было изменить. Например, если один из операндов команды находится в памяти, то другой может и не находиться в памяти. Следовательно, существуют команды сложения двух регистров, команды прибавления регистра к слову из памяти и команды прибавления слова из памяти к регистру, но не существует команд для прибавления одного слова памяти к другому слову памяти.

В первых архитектурах Intel все коды операций были размером 1 байт, хотя для изменения некоторых команд часто использовался так называемый префиксный

байт. **Префиксный байт** — это дополнительный код операции, который ставится перед командой, чтобы изменить ее действие. Примером префиксного байта может служить команда WIDE в машинах IJVM и JVM. К сожалению, в какой-то момент компания Intel вышла за пределы кодов операций, и один код операции, 0xFF, определялся как **код смены алфавита** и использовался для разрешения второго байта команды.

Отдельные биты в кодах операций процессора Pentium II дают довольно мало информации о команде. Единственной структурой такого рода в поле кода операции является младший бит в некоторых командах, который указывает, что именно вызывается — слово или байт, а также соседний бит, который указывает, является ли адрес памяти (если он вообще есть) источником или пунктом назначения. Таким образом, в большинстве случаев код операции должен быть полностью декодирован, чтобы установить, к какому классу относится операция, которую нужно выполнить, и, следовательно, какова длина этой команды. Это очень сильно снижает производительность, поскольку необходимо производить декодирование еще до того, как будет определено, где начинается следующая команда.

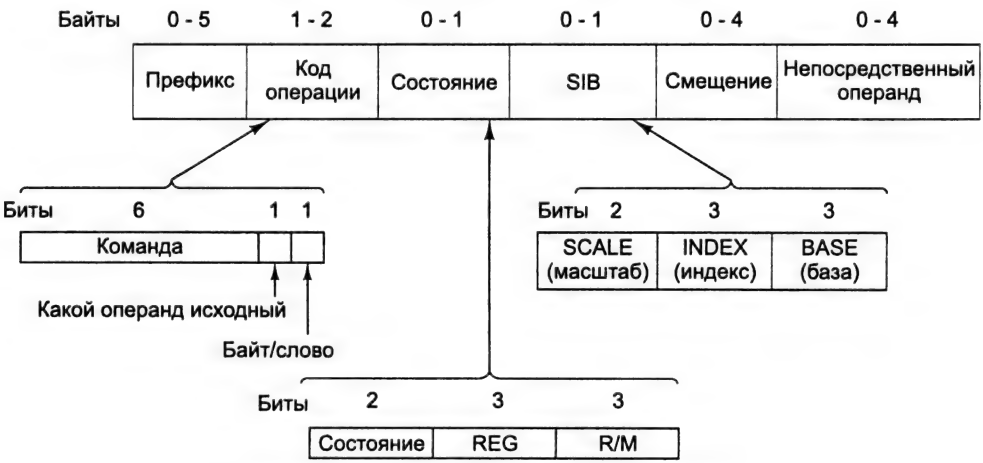


Рис. 5.9. Форматы команд процессора Pentium II

В большинстве команд вслед за байтом кода операции, который указывает местонахождение операнда в памяти, следует второй байт, который сообщает всю информацию об операнде. Эти 8 битов распределены в 2-битном поле MOD и двух 3-битных регистровых полях REG и R/M. Иногда первые три бита этого байта используются в качестве расширения для кода операции, давая в сумме 11 битов для кода операции. Тем не менее 2-битное поле означает, что существует только 4 способа обращения к операндам, и один из операндов всегда должен быть регистром. Логически должен быть определяем любой из регистров EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, но правила кодирования команд запрещают некоторые комбинации, поскольку эти комбинации используются для особых случаев. В некоторых типах команд требуется дополнительный байт, называемый **SIB (Scale, Index, Base — масштаб, индекс, база)**, который дает дополнительную спецификацию. Эта схема не идеальна, но она является компромиссом между требованием

обратной совместимости и желанием добавлять новые особенности, которые не были предусмотрены изначально.

Добавим еще, что некоторые команды имеют 1, 2 или 4 дополнительных байта для определения адреса команды (смещение), а иногда еще 1, 2 или 4 байта, содержащих константу (непосредственный операнд).

Форматы команд процессора UltraSPARC II

Архитектура команд процессора UltraSPARC II состоит из 32-битных команд, выровненных в памяти. Команды очень просты. Каждая из них определяет только одно действие. Типичная команда указывает два регистра, из которых поступают входные операнды, и один выходной регистр. Вместо одного из регистров команда может использовать константу со знаком. При выполнении команды LOAD два регистра (или один регистр и 13-битная константа) складываются вместе для определения адреса памяти, который нужно считать. Данные оттуда записываются в другой указанный регистр.

Изначально машина SPARC имела ограниченное число форматов команд. Они показаны на рис. 5.10. Со временем добавлялись новые форматы. Когда писалась эта книга, число форматов уже было равно 31. Большинство новых вариантов были получены путем отнимания нескольких битов из какого-нибудь поля. Например, изначально для команд перехода использовался формат 3 с 22-битным смещением. Когда были добавлены прогнозируемые переходы, 3 из 22 битов убирались: один из них стал использоваться для прогнозирования (совершать или не совершать переход), а два оставшихся определяли, какой набор битов условного кода нужно использовать. В результате получилось 19-битное смещение. Приведем другой пример. Существует много команд для переделывания одного типа данных в другой (целые числа в числа с плавающей точкой и т. д.). Для большинства этих команд используется вариант формата 1b, в котором поле непосредственной константы разбито на 5-битное поле, указывающее входной регистр, и 8-битное поле, которое обеспечивает дополнительные биты кода операции. Однако в большинстве команд все еще используются форматы, показанные на рисунке.

Первые два бита каждой команды помогают определить формат команды и сообщают аппаратному обеспечению, где найти оставшуюся часть кода операции, если она есть. В формате 1a оба источника операндов представляют собой регистры; в формате 1b один источник — регистр, а второй — константа в промежутке от −4096 до +4095. Бит 13 определяет один из этих двух форматов. (Биты нумеруются с 0.) В обоих случаях местом сохранения результатов всегда является регистр. Достаточный объем пространства обеспечен для 64 команд, некоторые из которых сохранены на будущее.

Поскольку все команды 32-битные, включить в команду 32-битную константу невозможно. Команда SETHI устанавливает 22 бита, оставляя пространство для другой команды, чтобы установить оставшиеся 10 битов. Это единственная команда, которая использует данный формат.

Для непрогнозируемых условных переходов используется формат 3, в котором поле УСЛОВИЕ определяет, какое условие нужно проверить. Бит A нужен для

того, чтобы избегать пустых операций при определенных условиях. Прогнозируемые переходы используют тот же формат, но только с 19-битным смещением, как было сказано выше.

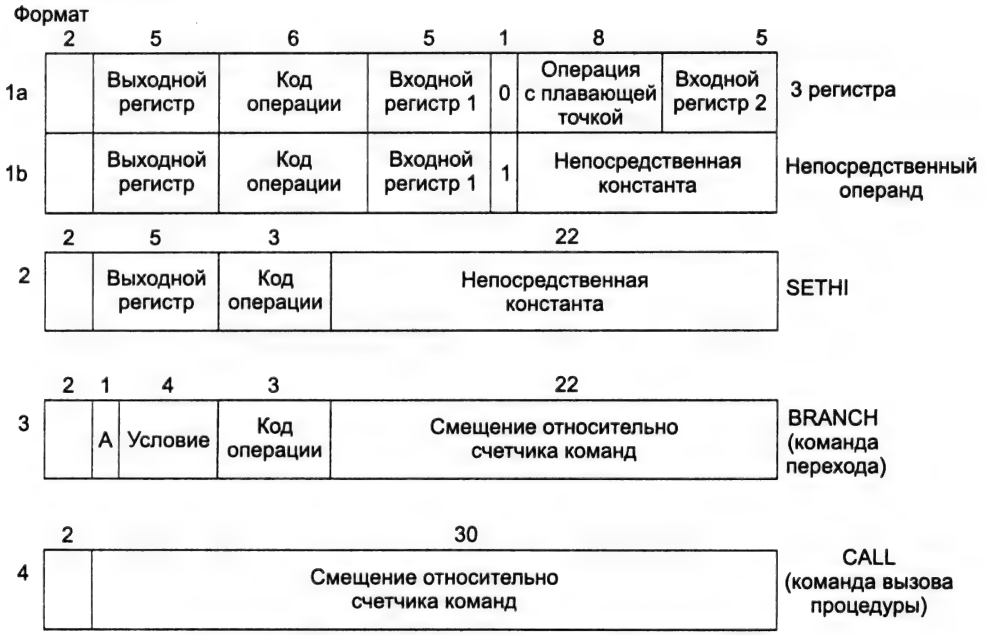


Рис. 5.10. Изначальные форматы команд процессора SPARC

Последний формат используется для команды вызова процедуры (`CALL`). Эта команда особая, поскольку только в ней для определения адреса требуется 30 битов. В данной архитектуре существует один 2-битный код операции. Требуемый адрес — это целевой адрес, разделенный на четыре.

Форматы команд JVM

Большинство форматов команд машины JVM чрезвычайно просты. Все форматы показаны на рис. 5.11. Их простота объясняется тем, что машина JVM сравнительно новая. Но подождите 10 лет. Все команды начинаются с кода операции в 1 байт. В некоторых командах за кодом операции следует второй байт. Это может быть индекс (как в команде `ILOAD`), константа (как в команде `BIPUSH`) или указатель типа данных (как в команде `NEWARRAY`, которая создает одномерный массив указанного типа в «куче»). Третий формат по сути такой же, как и второй, только вместо 8-битной константы там присутствует 16-битная константа (как, например, у команд `WIDE ILOAD` или `GOTO`). Формат 4 используется только для команды IINC. Формат 5 используется только для команды `MULTINEWARRAY`, которая создает многомерный массив «в куче». Формат 6 нужен только для команды `INVOKEINTERFACE`, которая вызывает процедуру при определенных обстоятельствах. Формат 7 предназначен только для команды `WIDE IINC`, чтобы обеспечить 16-битный индекс и 16-битную

константу, которая прибавляется к выбранной переменной. Формат 8 применяется только для команд `WIDE GOTO` и `WIDE JSR`, чтобы осуществлять переходы на большие расстояния в памяти и вызовы определенных процедур. Последний формат используется только двумя командами, и обе эти команды нужны для реализации оператора языка Java switch. Таким образом, все команды JVM, за исключением восьми особых команд, используют простые и короткие форматы 1, 2 и 3.

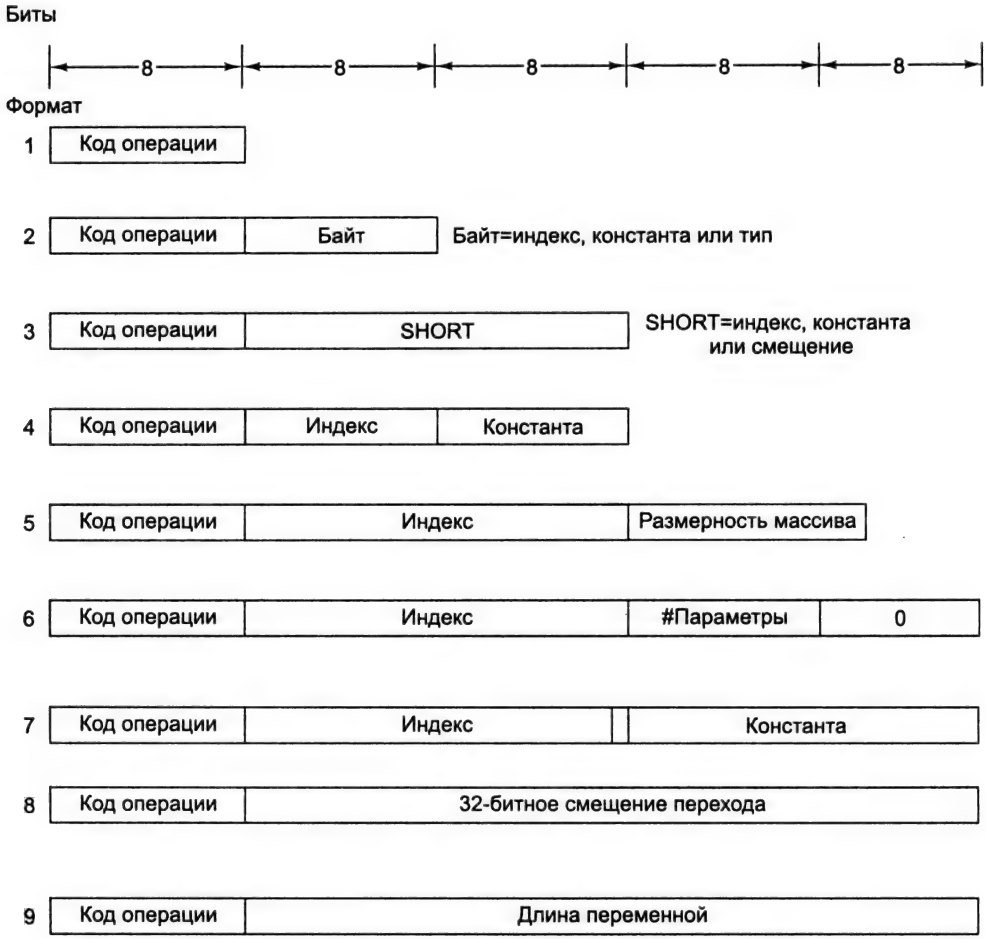


Рис. 5.11. Форматы команд JVM

В действительности дела обстоят гораздо лучше, чем может показаться. Команды виртуальной машины Java кодируются таким образом, чтобы большинство наиболее распространенных команд кодировались в одном байте. Большинство из 256 возможных команд, кодируемых в одном байте, представляют собой особые случаи общей формы команд IJVM.

Давайте, например, рассмотрим, как в машине Java происходит загрузка локальной переменной. Существует три различных способа определения локальной переменной. Самый короткий вариант покрывает самые распространенные случаи,

а самый длинный — все возможные случаи. JVM содержит команду ILOAD, использующую 8-битный индекс для определения локальной переменной, которую нужно поместить в стек. Мы также показали, как префикс WIDE позволяет использовать тот же код операции для определения любого из первых 65 536 элементов во фрейме локальных переменных. Для команды WIDE ILOAD требуется 4 байта: 1 — для WIDE, 1 — для ILOAD и 2 — для 16-битного индекса. Такое разделение объясняется тем, что большинство команд ILOAD используют одну из первых 256 локальных переменных. Префикс WIDE нужен для универсальности, применимости к любым ситуациям, и используется он редко.

Но разработчики машины JVM пошли еще дальше. Так как параметры процедуры передаются в первые несколько слов фрейма локальных переменных, команда ILOAD чаще всего использует элементы фрейма локальных переменных с невысокими индексами. Разработчики JVM решили, что стоит назначить отдельные 1-байтные коды операций для каждой из возможных комбинаций. Команда ILOAD_0 помещает в стек локальную переменную 0. Эта команда полностью эквивалентна 2-байтной команде ILOAD 0, за исключением того, что она занимает один байт вместо двух. Точно так же команды ILOAD_1, ILOAD_2 и ILOAD_3 (коды операций 0x1B, 0x1C и 0x1D соответственно) помещают в стек локальные переменные 1, 2 и 3. Отметим, что локальную переменную 1, например, можно загрузить одним из трех способов: ILOAD_1, ILOAD 1 и WIDE ILOAD 1.

Многие команды имеют варианты, подобные этим. Существуют специальные команды, полностью эквивалентные BIPUSH, для значений 0, 1, 2, 3, 4, 5, а также –1. Есть, кроме того, особые команды для записи переменных из стека в первые 4 слова пространства локальных переменных.

Отметим, что эти варианты повлекли за собой некоторые убытки. Только 256 различных команд могут определяться в одном байте. Поскольку 4 из этих 256 команд решили отвести на загрузку первых четырех локальных переменных, число команд уменьшилось на 4. Эти команды вместе с основной командой ILOAD в сумме составляют 5 команд. Префикс WIDE тоже использует одно из 256 возможных значений (а это даже не команда, а просто префикс), но он применяется к различным кодам операций.

Для спецификации загрузки операндов из набора констант разработчики использовали немного другой метод, поскольку они ожидали различия в распределении адресов. Они предоставили две версии команды: LDC и LDC_W. Вторая форма команды (она была включена в IJVM) может вызывать любое из 65 536 слов в наборе констант. В первой форме требуется только однобайтный индекс, но такая команда может вызывать только одно их первых 256 слов. Вызов любого из первых 256 слов можно осуществить с помощью 2-байтной команды, а вызов любого слова — с помощью 3-байтной команды. На эти 2 варианта требуется 2 кода из 256 кодов операций. Набор команд был бы более простым и регулярным, если бы разработчики выбрали ту же технологию, которую они использовали для команды ILOAD, то есть использовали бы префикс WIDE, а не команду LDC_W. Однако в этом случае для вызова констант из верхних 256 слов потребовалось бы 4 байта, а не 3.

Технология объединения кодов операций и индексов в один байт, а также размещения 256 доступных байтов в соответствии с частотой их использования была впервые предложена автором данной книги в 1978 году, но нашла применение только спустя два десятилетия [145].

Адресация

Разработка кодов операций является важной частью архитектуры команд. Однако значительное число битов программы используется для того, чтобы определить, откуда нужно брать операнды, а не для того, чтобы узнать, какие операции нужно выполнить. Рассмотрим команду ADD, которая требует спецификации трех операндов: двух источников и одного пункта назначения. (Термин «операнд» обычно используется применительно ко всем трем элементам, хотя пункт назначения — это место, где сохраняется результат.) Так или иначе команда ADD должна сообщать, где найти операнды и куда поместить результат. Если адреса памяти 32-битные, то спецификация этой команды требует помимо кода операции еще три 32-битных адреса. Адреса занимают гораздо больше бит, чем коды операции.

Два специальных метода предназначены для уменьшения размера спецификации. Во-первых, если операнд должен использоваться несколько раз, его можно переместить в регистр. В использовании регистра для переменной есть двойная польза: скорость доступа увеличивается, а для определения операнда требуется меньшее количество битов. Если имеется 32 регистра, любой из них можно определить, используя всего лишь 5 битов. Если при выполнении команды ADD применять только регистровые операнды, для определения всех трех операндов понадобится только 15 битов, а если бы эти операнды находились в памяти, понадобилось бы целых 96 битов.

Однако использование регистров может вызвать другую проблему. Если операнд, находящийся в памяти, должен сначала загружаться в регистр, то потребуется большее число битов для определения адреса памяти. Во-первых, для переноса операнда в регистр нужна команда LOAD. Для этого требуется не только код операции, но и полный адрес памяти, а также нужно определить целевой регистр. Поэтому если операнд используется только один раз, помещать его в регистр не стоит.

К счастью, многочисленные измерения показали, что одни и те же операнды используются многократно. Поэтому большинство новых архитектур содержат большое количество регистров, а большинство компиляторов доходят до огромных размеров, чтобы хранить локальные переменные в этих регистрах, устраняя таким образом многочисленные обращения к памяти. Это сокращает и размер, и время выполнения программы.

Второй метод подразумевает определение одного или нескольких операндов неявным образом. Для этого существует несколько технологий. Один из способов — использовать одну спецификацию для входного и выходного операндов. В то время как обычная трехадресная команда ADD использует форму

```
DESTINATION=SOURSE1+SOURSE2.
```

двухадресную команду можно сократить до формы

```
REGISER2=REGISTER2+SOURSE1.
```

Недостаток этой команды состоит в том, что содержимое REGISTER2 не сохранится. Если первоначальное значение понадобится позднее, его нужно сначала скопировать в другой регистр. Компромисс здесь заключается в том, что двухадресные команды короче, но они не так часто используются. У разных разработчи-

ков разные предпочтения. В Pentium II, например, используются двухадресные команды, а в UltraSPARC II — трехадресные.

Мы сократили число операндов команды ADD с трех до двух. Продолжим сокращение дальше. Первые компьютеры имели только один регистр, который назывался аккумулятором. Команда ADD, например, всегда прибавляла слово из памяти к аккумулятору, поэтому нужно было определять только один операнд (операнд памяти). Эта технология хорошо работала для простых вычислений, но когда были нужны промежуточные результаты, аккумулятор приходилось записывать обратно в память, а позднее вызывать снова. Следовательно, эта технология нам не подходит.

Итак, мы перешли от трехадресной команды ADD к двухадресной, а затем к одноадресной. Что же остается? Ноль адресов? Да. В главе 4 мы увидели, как машина IJVM использует стек. Команда IADD не имеет адресов. Входные и выходные операнды не показываются явным образом. Ниже мы рассмотрим стековую адресацию более подробно.

Способы адресации

До сих пор мы не рассказывали о том, как интерпретируются биты адресного поля для нахождения операнда. Один из возможных вариантов состоит в том, что они содержат адрес операнда. Помимо огромного поля, необходимого для определения полного адреса памяти, данный метод имеет еще одно ограничение: этот адрес должен определяться во время компиляции. Существуют и другие возможности, которые обеспечивают более короткие спецификации, а также могут определять адреса динамически. В следующих разделах мы рассмотрим некоторые из этих форм, которые называются **способами адресации**.

Непосредственная адресация

Самый простой способ определения операнда — содержать в адресной части сам операнд, а не адрес операнда или какую-либо другую информацию, описывающую, где находится операнд. Такой операнд называется **непосредственным операндом**, поскольку он автоматически вызывается из памяти одновременно с командой; следовательно, он сразу непосредственно становится доступным. Один из вариантов команды с непосредственным адресом для загрузки в регистр R1 константы 4 показан на рис. 5.12.

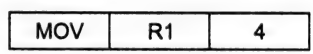


Рис. 5.12. Команда с непосредственным адресом для загрузки константы 4 в регистр 1

При непосредственной адресации не требуется дополнительного обращения к памяти для вызова операнда. Однако у такого способа адресации есть и некоторые недостатки. Во-первых, таким способом можно работать только с константами. Во-вторых, число значений ограничено размером поля. Тем не менее эта технология используется во многих архитектурах для определения целочисленных констант.

Прямая адресация

Следующий способ определения операнда — просто дать его полный адрес. Такой способ называется **прямой адресацией**. Как и непосредственная адресация, прямая адресация имеет некоторые ограничения: команда всегда будет иметь доступ только к одному и тому же адресу памяти. То есть значение может меняться, а адрес — нет. Таким образом, прямая адресация может использоваться только для доступа к глобальным переменным, адреса которых известны во время компиляции. Многие программы содержат глобальные переменные, поэтому этот способ широко используется. Каким образом компьютер узнает, какие адреса непосредственные, а какие прямые, мы обсудим позже.

Регистровая адресация

Регистровая адресация по сути сходна с прямой адресацией, только в данном случае вместо ячейки памяти определяется регистр. Поскольку регистры очень важны (из-за быстрого доступа и коротких адресов), этот способ адресации является самым распространенным на большинстве компьютеров. Многие компиляторы доходят до огромных размеров, чтобы определить, к каким переменным доступ будет осуществляться чаще всего (например, индекс цикла), и помещают эти переменные в регистры.

Такой способ адресации называют **регистровой адресацией**. В архитектурах с загрузкой с запоминанием, например UltraSPARC II, практически все команды используют исключительно этот способ адресации. Он не используется только в том случае, когда операнд перемещается из памяти в регистр (команда `LOAD`) или из регистра в память (команда `STORE`). Даже в этих командах один из операндов является регистром — туда отправляется слово из памяти или оттуда перемещается слово в память.

Косвенная регистровая адресация

При таком способе адресации определяемый операнд берется из памяти или отправляется в память, но адрес не зафиксирован жестко в команде, как при прямой адресации. Вместо этого адрес содержится в регистре. Если адрес используется таким образом, он называется **указателем**. Преимущество косвенной адресации состоит в том, что можно обращаться к памяти, не имея в команде полного адреса. Кроме того, при разных выполнениях данной команды можно использовать разные слова памяти.

Чтобы понять, почему может быть полезно использование разных слов при каждом выполнении команды, представим себе цикл, который проходит по 1024-элементному одномерному массиву целых чисел для вычисления суммы элементов в регистре R1. Вне этого цикла какой-то другой регистр, например R2, может указывать первый элемент массива, а еще один регистр, например R3, может указывать первый адрес после массива. Массив содержит 1024 целых числа по 4 байта каждое. Если массив начинается с A, то первый адрес после массива будет A+4096. Типичная программа ассемблера, выполняющая это вычисление для двухадресной машины, показана в листинге 5.1.

Листинг 5.1. Программа на ассемблере для вычисления суммы элементов массива

```
        MOV R1,#0          ;накопление суммы в R1, изначально 0
        MOV R2,#A          ;R2=адрес массива A
        MOV R3,#A+4096     ;R3=адрес первого слова после A
LOOP:   ADD R1,(R2)        ;получение операнда через регистр R2
        ADD R2,#4          ;увеличение R2 на одно слово(4байта)
        CMP R2,R3          ;проверка на завершение
        BLT LOOP           ;если R2<R3, продолжать цикл
```

В этой маленькой программе мы использовали несколько способов адресации. Первые три команды используют регистровую адресацию для первого операнда (пункт назначения) и непосредственную адресацию для второго операнда (константа, обозначенная символом #). Вторая команда помещает в R2 не содержимое A, а *адрес* A. Именно это и сообщает ассемблеру знак #. Сходным образом третья команда помещает в R3 первое слово после массива.

Интересно отметить, что само тело цикла не содержит каких-либо адресов памяти. В четвертой команде используется регистровая и косвенная адресация. В пятой команде применяется регистровая и непосредственная адресация, а в шестой — два раза регистровая. Команда `BLT` могла бы использовать адрес памяти, однако более привлекательным является определение адреса с помощью 8-битного смещения, связанного с самой командой `BLT`. Таким образом, полностью избегая адресов памяти, мы получили короткий и быстрый цикл. Кстати, эта программа предназначена для Pentium II, только мы переименовали команды и регистры и для упрощения понимания изменили запись.

Теоретически есть еще один способ выполнения этого вычисления без использования косвенной регистровой адресации. Этот цикл мог бы содержать команду для прибавления A к регистру R1, например

```
ADD R1,A
```

Тогда при каждом шаге команда должна увеличиваться на 4. Таким образом, после одного шага команда будет выглядеть следующим образом:

```
ADD R1,A+4
```

и так далее до завершения цикла.

Программа, которая сама изменяется подобным образом, называется **самоизменяющейся программой**. Эта идея была предложена Джоном фон Нейманом и применялась в старых компьютерах, где не было косвенной регистровой адресации. В настоящее время самоизменяющиеся программы считаются неудобными и очень трудными для понимания. Кроме того, их выполнение нельзя разделить между несколькими процессорами. Они даже не могут правильно выполняться на машинах с разделенной кэш-памятью первого уровня, если в кэш-памяти команд нет специальной схемы для обратной записи (поскольку разработчики предполагали, что программы сами себя не изменяют).

Индексная адресация

Часто нужно уметь обращаться к словам памяти по известному смещению. Подобные примеры мы видели в машине IJVM, где локальные переменные определяются по смещению от регистра LV. Обращение к памяти по регистру и константе смещения называется **индексной адресацией**.

В машине IJVM при доступе к локальной переменной используется указатель ячейки памяти (LV) в регистре плюс небольшое смещение в самой команде, как показано на рис. 4.14, *а*. Есть и другой способ: указатель ячейки памяти в команде и небольшое смещение в регистре. Чтобы показать, как это работает, рассмотрим следующий пример. У нас есть два одномерных массива A и B по 1024 слова в каждом. Нам нужно вычислить A_i И B_i для всех пар, а затем соединить все эти 1024 логических произведения операцией ИЛИ, чтобы узнать, есть ли в этом наборе хотя бы одна пара, не равная нулю. Один из вариантов — поместить адрес массива A в один регистр, а адрес массива B — в другой регистр, а затем последовательно перебирать элементы массивов, аналогично тому, как мы делали в предыдущей программе (см. листинг 5.1). Такая программа, конечно же, будет работать, но ее можно усовершенствовать, как показано в листинге 5.2.

Листинг 5.2. Программа на языке ассемблера для вычисления операции ИЛИ от (Ai И Bi) для массива из 1024 элементов

```
        MOV R1,#0       ;собирает результаты выполнения ИЛИ в R1.
        MOV R2,#0       ;R2= i от текущего произведения A[i] И B[i]
        MOV R3,#4096    ;R3=первое ненужное значение индекса

LOOP:   MOV R4,A(R2)    ;R4=A[i]
        AND R4,B(R2)    ;R4=A[i] И B[i]
        OR R1,R4
        ADD R2,#4       ;i=i+4
        CMP R2,R3       ;нужно ли продолжать?
        BLT LOOP        ;если R2<R3, мы не закончили и нужно продолжать
```

Здесь нам требуется 4 регистра:

1. R1 — содержит результаты суммирования логических произведений.
2. R2 — индекс i, который используется для перебора элементов массива.
3. R3 — константа 4096. Это самое маленькое значение i, которое не используется.
4. R4 — временный регистр для хранения каждого произведения.

После инициализации регистров мы входим в цикл из шести команд. Команда напротив LOOP вызывает элемент Ai в регистр R4. При вычислении источника здесь используется индексная адресация. Регистр (R2) и константа (адрес элемента A) складываются, и полученный результат используется для обращения к памяти. Сумма этих двух величин поступает в память, но не сохраняется ни в одном из видимых пользователем регистров. Запись

```
MOV R4,A(R2)
```

означает, что для определения пункта назначения используется регистровая адресация, где R4 — это регистр, а для определения источника используется индексная адресация, где A — это смещение, а R2 — это регистр. Если A имеет значение, скажем, 124300, то соответствующая машинная команда будет выглядеть так, как показано на рис. 5.13.

MOV	R4	R2	124300

Рис. 5.13. Возможное представление команды MOV R4, A(R2)

Во время первого прохождения цикла регистр R2 принимает значение 0 (поскольку регистр инициализируется таким образом), поэтому нужное нам слово A0 находится в ячейке с адресом 124300. Это слово загружается в регистр R4. При следующем прохождении цикла R2 принимает значение 4, поэтому нужное нам слово A1 находится в ячейке с адресом 124304 и т. д.

Как мы говорили раньше, здесь смещение — это указатель ячейки памяти, а значение регистра — это небольшое целое число, которое во время вычисления меняется. Такая форма требует, чтобы поле смещения в команде было достаточно большим для хранения адреса, поэтому такой способ не очень эффективен. Тем не менее этот способ часто оказывается самым лучшим.

Относительная индексная адресация

В некоторых машинах применяется способ адресации, при котором адрес вычисляется путем суммирования значений двух регистров и смещения (смещение факультативно). Такой подход называется **относительной индексной адресацией**. Один из регистров — это база, а другой — это индекс. Такая адресация очень удобна при следующей ситуации. Вне цикла мы могли бы поместить адрес элемента A в регистр R5, а адрес элемента B в регистр R6. Тогда мы могли бы заменить две первые команды цикла LOOP на

```
LOOP:    MOV R4,(R2+R5)
AND R4,(R2+R6)
```

Было бы идеально, если бы существовал способ адресации по сумме двух регистров без смещения. С другой стороны, даже команда с 8-битным смещением была бы большим достижением, поскольку мы оба смещения могли бы установить на 0. Однако если смещения всегда составляют 32 бита, тогда мы ничего не выиграем, используя такую адресацию. На практике машины с такой адресацией обычно имеют форму с 8-битным и 16-битным смещением.

Стековая адресация

Мы уже говорили, что очень желательно сделать машинные команды как можно короче. Конечный предел в сокращении длины адреса — это команды без адресов. Как мы видели в главе 4, безадресные команды, например IADD, возможны при наличии стека. В этом разделе мы рассмотрим стековую адресацию более подробно.

Обратная польская запись

В математике существует древняя традиция помещать оператор между операндами (x+y), а не после операндов (xy+). Форма с оператором между операндами называется **инфиксной записью**. Форма с оператором после операндов называется **постфиксной** или **обратной польской записью** в честь польского логика Я. Лукасевича (1958), который изучал свойства этой записи.

Обратная польская запись имеет ряд преимуществ над инфиксной записью для выражения алгебраических формул. Во-первых, любая формула может быть выражена без скобок. Во-вторых, она удобна для вычисления формул в машинах со

стеками. В-третьих, инфиксные операторы имеют приоритеты, которые произвольны и нежелательны. Например, мы знаем, что a×b+c значит (a×b)+c, а не a×(b+c), поскольку произвольно было определено, что умножение имеет приоритет над сложением. Но имеет ли приоритет сдвиг влево над логической операцией И? Кто знает? Обратная польская запись устраняет такие недоразумения.

Существует несколько алгоритмов для превращения инфиксных формул в обратную польскую запись. Ниже изложена переделка идеи Э. Дейкстры. Предположим, что формула состоит из следующих символов: переменных, двухоперандных операторов +, –, *, /, а также левой и правой скобок. Чтобы отметить конец формулы, мы будем вставлять символ ⊥ после последнего символа одной формулы и перед первым символом следующей формулы.

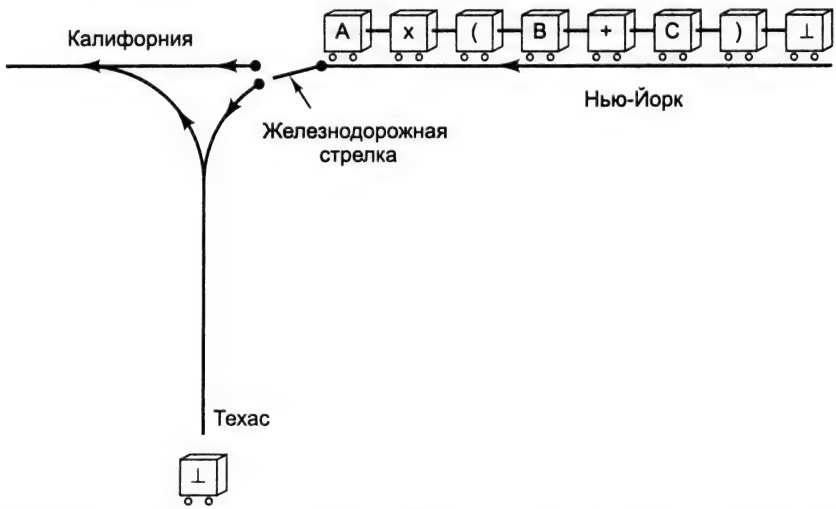


Рис. 5.14. Каждый вагон представляет собой один символ в формуле, которую нужно переделать из инфиксной формы в обратную польскую запись

На рис. 5.14 нарисована железная дорога из Нью-Йорка в Калифорнию с развилкой, ведущей в Техас. Каждый символ формулы представлен одним вагоном. Поезд движется на запад (налево). Перед развилкой каждый вагон должен останавливаться и узнавать, должен ли он двигаться прямо в Калифорнию, или ему нужно по пути заехать в Техас. Вагоны, содержащие переменные, всегда направляются Калифорнию и никогда не едут в Техас. Вагоны, содержащие все прочие символы, должны перед вхождением на развилку узнавать о содержимом ближайшего вагона, отправившегося в Техас.

В таблице на рис. 5.15 показана зависимость ситуации от того, какой вагон отправился последним в Техас и какой вагон находится у развилки. Первый ⊥ всегда отправляется в Техас. Числа соответствуют следующим ситуациям:

1. Вагон на развилке направляется в Техас.
2. Последний вагон, направившийся в Техас, разворачивается и направляется в Калифорнию.
3. Вагон, находящийся на развилке, и последний вагон, отправившийся в Техас, угоняются и исчезают (то есть оба удаляются).

4. Остановка. Символы, находящиеся в Калифорнии, представляют собой формулу в обратной польской записи, если читать слева направо.
5. Остановка. Произошла ошибка. Изначальная формула была некорректно сбалансирована.



Рис. 5.15. Алгоритм преобразования инфиксной записи в обратную польскую запись

После каждого действия производится новое сравнение вагона, находящегося у развилки (это может быть тот же вагон, что и в предыдущем сравнении, а может быть следующий вагон), и вагона, который на данный момент последним ушел на Техас. Этот процесс продолжается до тех пор, пока не будет достигнут шаг 4. Отметим, что линия на Техас используется как стек, где отправка вагона в Техас — это помещение элемента в стек, а разворот вагона, отправленного в Техас, в сторону Калифорнии — это выталкивание элемента из стека.

Таблица 5.5. Некоторые примеры инфиксных выражений и их эквиваленты в обратной польской записи

Инфиксная запись	Обратная польская запись
A+B×C	A B C ×+
A×B+C	A B× C+
A×B+C×D	A B× C D×+
(A+B)/(C–D)	A B+C D–/
A×B/C	A B × C/
((A+B) ×C+D)/(E+F+G)	A B+C×D+ E F + G +/

Порядок переменных в инфиксной и обратной польской записи одинаков. Однако порядок операторов не всегда один и тот же. В обратной польской записи операторы появляются в том порядке, в котором они будут выполняться. В табл. 5.5 даны примеры инфиксных формул и их эквивалентов в обратной польской записи.

Вычисление формул в обратной польской записи

Обратная польская запись — идеальная запись для вычисления формул на компьютере со стеком. Формула состоит из n символов, каждый из которых является или операндом, или оператором. Алгоритм для вычисления формулы в обратной

польской записи с использованием стека прост. Нужно просто прочитать обратную польскую запись слева направо. Если встречается операнд, его нужно поместить в стек. Если встречается оператор, нужно выполнить соответствующую команду.

В таблице 5.6 показано вычисление выражения

(8+2×5) / (1+3×2−4)

в машине JVM. Соответствующая формула в обратной польской записи выглядит следующим образом:

825×+132×+4−/

В таблице мы ввели команды умножения и деления `IMUL` и `IDIV`. Число на вершине стека — это правый операнд (а не левый). Это очень важно для операций деления и вычитания, поскольку порядок операндов в данном случае имеет значение (в отличие от операций сложения и умножения). Другими словами, команда `IDIV` определяется следующим образом: сначала в стек помещается числитель, потом знаменатель, и тогда выполнение операции дает правильный результат. Отметим, что преобразовать обратную польскую запись в код (I)JVM очень легко: нужно просто просканировать формулу в обратной польской записи и выдавать одну команду с каждым символом. Если символ является константой или переменной, нужно выдавать команду помещения этой константы или переменной в стек. Если символ является оператором, нужно выдавать команду для выполнения данной операции.

Способы адресации для команд перехода

До сих пор мы рассматривали только те команды, которые оперируют с данными. Командам перехода (а также командам вызова процедур) также нужны особые способы адресации для определения целевого адреса. Способы, о которых мы говорили в предыдущих разделах, работают и для большинства команд перехода. Один из возможных вариантов — прямая адресация, когда целевой адрес просто полностью включается в команду.

Другие способы адресации тоже имеют смысл. Косвенная регистровая адресация позволяет программе вычислять целевой адрес, помещать его в регистр, а затем переходить туда. Такой способ дает максимальную гибкость, поскольку целевой адрес вычисляется во время выполнения программы. Но он также предоставляет огромные возможности для появления ошибок, которые практически невозможно найти.

Индексная адресация, при которой известно смещение от регистра, также является вполне разумным способом. Этот способ обладает теми же свойствами, что и косвенная регистровая адресация.

Еще один вариант — относительная адресация по счетчику команд. В данном случае для получения целевого адреса смещение (со знаком), находящееся в самой команде, прибавляется к программному счетчику. По сути, это индексная адресация, где в качестве регистра используется PC.

Таблица 5.6. Использование стека для вычисления формулы в обратной польской записи

Шаг	Оставшаяся цепочка	Команда	Стек
1	8 2 5 × + 1 3 2 × + 4 – /	BIPUSH 8	8
2	2 5 × + 1 3 2 × + 4 – /	BIPUSH 2	8, 2
3	5 × + 1 3 2 × + 4 – /	BIPUSH 5	8, 2, 5
4	× + 1 3 2 × + 4 – /	IMUL	8, 10
5	+ 1 3 2 × + 4 – /	IADD	18
6	1 3 2 × + 4 – /	BIPUSH 1	18, 1
7	3 2 × + 4 – /	BIPUSH 3	18, 1, 3
8	2 × + 4 – /	BIPUSH 2	18, 1, 3, 2
9	× + 4 – /	IMUL	18, 1, 6
10	+ 4 – /	IADD	18, 7
11	4 – /	BIPUSH 4	18, 7, 4
12	– /	ISUB	18, 3
13	/	IDIV	6

Ортогональность кодов операций и способов адресации

С точки зрения программного обеспечения, команды и способы адресации должны иметь регулярную структуру, число форматов команд должно быть минимальным. При такой структуре компилятору гораздо проще порождать нужный код. Все коды операций должны допускать все способы адресации, где это имеет смысл. Более того, все регистры должны быть доступны для всех типов регистров (включая указатель фрейма (FP), указатель стека (SP) и программный счетчик (PC)).

Рассмотрим 32-битные форматы команд для трехадресной машины (рис. 5.16). Здесь поддерживаются до 256 кодов операций. В формате 1 каждая команда имеет два входных регистра и один выходной регистр. Этот формат используется для всех арифметических и логических команд.

Неиспользованное 6-битное поле в конце формата может использоваться для дальнейшей дифференциации команд. Например, можно иметь один код для всех операций с плавающей точкой, а различаться эти операции будут по дополнительному полю. Кроме того, если установлен бит 23, тогда используется формат 2, а второй операнд уже не является регистром, а является 13-битной непосредственной константой со знаком. Команды LOAD и STORE тоже могут использовать этот формат для обращения к памяти при индексном способе адресации.

Необходимо также иметь небольшое число дополнительных команд (например, команды условных переходов), но они легко подходят под формат 3. Например, можно приписать один код операции каждому (условному) переходу, вызову процедуры и т. д., тогда останется 24 бита для смещения по счетчику команд. Если предположить, что это смещение считается в словах, период будет составлять ± 32 Мбайт. Несколько кодов операций можно зарезервировать для команд LOAD

и STORE, которым нужны длинные смещения из формата 3. Они не будут общими (например, только регистр R0 будет загружаться и сохраняться), и использоваться будут довольно редко.



Рис. 5.16. Разработка форматов команд для трехадресной машины

Теперь рассмотрим разработку для двухадресной машины, в которой в качестве любого операнда может использоваться слово из памяти (рис. 5.17). Такая машина может прибавлять слово из памяти к регистру, прибавлять регистр к слову из памяти, складывать два регистра или складывать два слова из памяти. В настоящее время осуществлять доступ к памяти довольно дорого, поэтому данный проект не очень популярен, но если с развитием технологий доступ к памяти в будущем станет дешевле, такой подход будет считаться простым и эффективным. Машины PDP-11 и VAX были очень популярны и доминировали на рынке мини-компьютеров в течение двух десятилетий. В этих машинах использовались форматы, сходные с тем, который изображен на рис. 5.17.

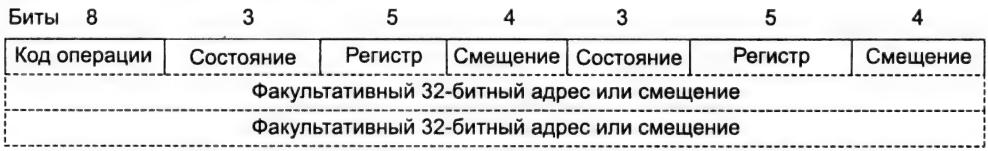


Рис. 5.17. Разработка форматов команд для двухадресной машины

Здесь мы снова имеем 8-битный код операции, но теперь у нас есть 12 битов для определения источника и 12 битов для определения пункта назначения. Для каждого операнда 3 бита дают метод адресации, 5 битов дают регистр и 4 бита дают смещение. Имея 3 бита для установления метода адресации, мы можем поддерживать непосредственную, прямую, регистровую, косвенную регистровую индексную и стековую адресации, и при этом еще остается место для двух дополнительных методов, которые, возможно, появятся в будущем. Это простая разработка, которую легко компилировать; она достаточно гибкая, особенно если счетчик программ, указатель стека и указатель локальных переменных находятся среди регистров общего назначения, к которым можно получить доступ.

Единственная проблема, которая здесь есть, — это то, что при прямой адресации нам нужно большее количество битов для адреса. В машинах PDP-11 и VAX к

команде было добавлено дополнительное слово для адреса каждого прямо адресуемого операнда. Мы тоже могли бы использовать один из двух доступных способов адресации для индексной адресации с 32-битным смещением, которое следует за командой. Тогда в худшем случае при прибавлении слова из памяти к слову из памяти, когда обращение к обоим операндам производится с помощью прямой адресации, или при использовании длинной индексной формы команда была бы 96 битов в длину и занимала бы 3 цикла шины (один — для команды и два — для данных). С другой стороны, большинству разработок типа RISC потребовалось бы по крайней мере 96 битов, а может и больше, для прибавления произвольного слова из памяти к другому произвольному слову из памяти, и тогда нужно было бы по крайней мере 4 цикла шины.

Помимо форматов, изображенных на рис. 5.17, возможны и другие варианты. В данной разработке можно выполнять операцию

```
i=j;
```

с помощью одной 32-битной команды, при условии что i и j находятся среди первых 16 локальных переменных. Для переменных после 16 нам приходится переходить к 32-битным смещениям. Можно сделать другой формат с одним 8-битным смещением вместо двух 4-битных и правилом, что это смещение может использоваться либо источником, либо пунктом назначения, но не тем и другим одновременно. Варианты компромиссов не ограничены, и разработчики должны учитывать многие факторы, чтобы получить хороший результат.

Способы адресации процессора Pentium II

Способы адресации процессора Pentium II чрезвычайно нерегулярны и зависят от того, в каком формате находится конкретная команда — 16-битном или 32-битном. Мы не будем рассматривать 16-битные команды. Вполне достаточно 32-битных. Поддерживаемые способы адресации включают непосредственную, прямую, регистровую, косвенную регистровую индексную и специальную адресацию для обращения к элементам массива. Проблема заключается в том, что не все способы применимы ко всем командам и не все регистры могут использоваться при всех способах адресации. Это сильно усложняет работу составителя компилятора.

Байт MODE на рис. 5.9 управляет способами адресации. Один из операндов определяется по комбинации полей MOD и R/M. Второй операнд всегда является регистром и определяется по значению поля REG. В таблице 5.7 приведен список 32 комбинаций значений 2-битного поля MOD и 3-битного поля R/M. Например, если оба поля равны 0, операнд считывается из ячейки памяти с адресом, который содержится в регистре EAX.

Колонки 01 и 10 включают способы адресации, при которых значение регистра прибавляется к 8-битному или 32-битному смещению, которое следует за командой. Если выбрано 8-битное смещение, оно перед сложением получает 32-битное знаковое расширение. Например, команда `ADD` с полем R/M=011, полем MOD=01 и смещением, равным 6, вычисляет сумму регистра EBX и 6, и в качестве одного из операндов считывает слово из полученного адреса памяти. Значение регистра EBX не изменяется.

Таблица 5.7. 32-битные способы адресации процессора Pentium II. M[x] — это слово в памяти с адресом x

	MOD			
R/M	**00**	**01**	**10**	**11**
000	M[EAX]	M[EAX+СМЕЩЕНИЕ 8]	M[EAX+СМЕЩЕНИЕ 32]	EAX или AL
001	M[ECX]	M[ECX+СМЕЩЕНИЕ 8]	M[ECX+СМЕЩЕНИЕ 32]	ECX или CL
010	M[EDX]	M[EDX+СМЕЩЕНИЕ 8]	M[EDX+СМЕЩЕНИЕ 32]	EDX или DL
011	M[EBX]	M[EBX+СМЕЩЕНИЕ 8]	M[EBX+СМЕЩЕНИЕ 32]	EBX или BL
100	SIB	SIB и СМЕЩЕНИЕ 8	SIB и СМЕЩЕНИЕ 32	ESP или AH
101	Прямая адресация	M[EBP+СМЕЩЕНИЕ 8]	M[EBP+СМЕЩЕНИЕ 32]	EBP или CH
110	M[ESI]	M[ESI+СМЕЩЕНИЕ 8]	M[ESI+СМЕЩЕНИЕ 32]	ESI или DH
111	M[EDI]	M[EDI+СМЕЩЕНИЕ 8]	M[EDI+СМЕЩЕНИЕ 32]	EDI или BH

При MOD=11 предоставляется выбор из двух регистров. Для команд со словами берется первый вариант, для команд с байтами — второй. Отметим, что здесь не все регулярно. Например, нельзя осуществить косвенную адресацию через EBP или прибавить смещение к ESP.

Иногда вслед за байтом MODE следует дополнительный байт **SIB (Scale, Index, Base — масштаб, индекс, база)** (см. рис. 5.9). Байт SIB определяет масштабный коэффициент и два регистра. Когда присутствует байт SIB, адрес операнда вычисляется путем умножения индексного регистра на 1, 2, 4 или 8 (в зависимости от SCALE), прибавлением его к базовому регистру и, наконец, возможным прибавлением 8- или 32-битного смещения, в зависимости от значения поля MOD. Практически все регистры могут использоваться и в качестве индекса, и в качестве базы.

Форматы SIB могут пригодиться для обращения к элементам массива. Рассмотрим следующее выражение на языке Java:

```
for (i=0; i<n; i++) a[i]=0;
```

где a — это массив 4-байтных целых чисел, относящийся к текущей процедуре. Обычно регистр EBP используется для указания на базу стекового фрейма, который содержит локальные переменные и массивы, как показано на рис. 5.18. Компилятор должен хранить i в регистре EAX. Для доступа к элементу a[i] он будет использовать формат SIB, в котором адрес операнда равен сумме 4×EAX, EBP и 8. Эта команда может сохраняться в a[i] за одну команду.

А стоит ли применять такой способ адресации? На этот вопрос трудно ответить. Без сомнения, эта команда при надлежащем использовании сохраняет несколько циклов. Насколько часто она используется, зависит от компилятора и от приложения. Проблема здесь в том, что эта команда занимает определенное количество пространства микросхемы, которое можно было бы использовать для других целей, если бы этой команды не было. Например, можно было бы сделать больше кэш-память первого уровня.

Мы представили несколько возможных компромиссов, с которыми постоянно сталкиваются разработчики. Обычно перед тем как воплотить какую-либо идею в кремнии, производятся обширные моделирующие прогоны, но для этого нужно

иметь представление о том, какова рабочая нагрузка. Можно быть уверенным, что разработчики машины 8088 не включили web-браузер в набор тестов. Решения, принятые 20 лет назад, могут оказаться абсолютно неудачными с точки зрения современных приложений. Однако если какая-либо особенность была включена в машину, избавиться от нее уже невозможно по причине требования совместимости.

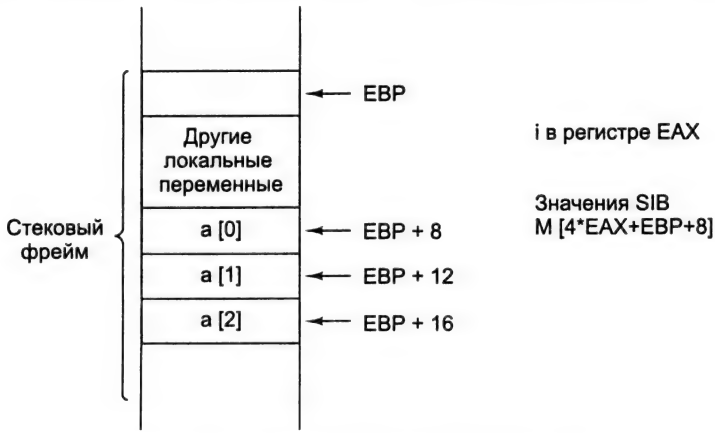


Рис. 5.18. Обращение к элементу массива a[i]

Способы адресации процессора UltraSPARC II

В архитектуре команд процессора UltraSPARC все команды используют непосредственную или регистровую адресацию, за исключением тех команд, которые обращаются к памяти. При регистровом способе адресации 5 битов просто сообщают, какой регистр нужно использовать. При непосредственной адресации данные обеспечивает 13-битная константа со знаком. Для арифметических, логических и подобных команд никакие другие способы адресации не используются.

К памяти обращаются команды трех типов: команды загрузки (LOAD), команды сохранения (STORE) и одна команда синхронизации мультипроцессора. Для команд LOAD и STORE есть два способа обращения к памяти. Первый способ: вычисляется сумма двух регистров, а затем через полученное значение производится косвенная адресация. Второй способ представляет собой обычное индексирование с 13-битным смещением со знаком.

Способы адресации машины JVM

У машины JVM нет общих способов адресации в том смысле, что каждая команда содержит несколько битов, которые сообщают, как нужно вычислить адрес (как в Pentium II, например). Вместо этого здесь с каждой командой связан один особый способ адресации. Поскольку в JVM нет видимых регистров, регистровая и косвенная регистровая адресация здесь невозможна. Несколько команд, например BIPUSH, используют непосредственную адресацию. Единственный оставшийся

доступный способ — индексная адресация. Она используется командами LOAD, ISTORE, LDC_W, а также несколькими командами, которые определяют переменную, связанную с каким-нибудь неявным регистром, обычно LV или CPP. Команды перехода тоже используют индексную адресацию, при этом PC рассматривается как регистр.

Сравнение способов адресации

Мы только что рассмотрели несколько способов адресации. Способы адресации машин Pentium II, UltraSPARC II и JVM изложены в табл. 5.8. Как мы уже говорили, не каждый способ может использоваться любой командой.

Таблица 5.8. Сравнение способов адресации

Способ адресации	Pentium II	UltraSPARC II	JVM
Непосредственная	×	×	×
Прямая	×		
Регистровая	×	×	
Косвенная регистровая	×		
Индексная	×	×	×
Относительная индексная		×	
Стековая			×

На практике для эффективной архитектуры команд вовсе не требуется большого количества различных способов адресации. Поскольку практически весь код, написанный на этом уровне, будет порождаться компиляторами, способов адресации должно быть мало, и они должны быть четкими и ясными. Машина должна предлагать либо все возможные варианты, либо только один вариант.

В остальных промежуточных случаях может оказаться так, что компилятор не способен сделать выбор.

Поэтому самые простые архитектуры используют очень небольшое число способов адресации, причем на каждый из этих способов накладываются жесткие ограничения. Обычно практически для любых применений достаточно непосредственной, прямой, регистровой и индексной адресации. Каждый регистр (включая указатель локальных переменных, указатель стека и счетчик программ) должен быть пригоден к употреблению всякий раз, когда этот регистр требуется. Более сложные способы адресации могут сократить число команд, но при этом придется ввести последовательности операций, которые трудно будет выполнять параллельно с другими последовательными операциями.

Мы рассмотрели возможные компромиссы между кодами операций и адресами и между различными способами адресации. Когда вы сталкиваетесь с новым компьютером, вы должны изучить все команды и способы адресации не только для того, чтобы знать, какие из них имеются в наличии, но и для того, чтобы понять, почему был сделан именно такой выбор и каковы были бы последствия при другом выборе.

Типы команд

Команды можно грубо поделить на несколько групп, которые повторяются от машины к машине, хотя и могут различаться в деталях. Кроме того, в каждом компьютере всегда имеется несколько необычных команд, которые добавлены в целях совместимости с предыдущими моделями или из-за того, что у разработчика возникла блестящая идея, или потому, что правительство заплатило производителю, чтобы тот включил эту команду в набор команд. Ниже мы попытаемся описать все наиболее распространенные категории. Отметим, что мы не претендуем на исчерпывающее изложение.

Команды перемещения данных

Копирование данных из одного места в другое — одна из самых распространенных операций. Под копированием мы понимаем создание нового объекта с точно таким же набором битов, как у исходного объекта. Такое понимание слова «перемещение» несколько отличается от его обычного значения. Если мы говорим, что какой-то человек переместился из Нью-Йорка в Калифорнию, это не значит, что в Калифорнии была создана идентичная копия этого человека, а оригинал остался в Нью-Йорке. Когда мы говорим, что содержимое ячейки памяти 2000 переместилось в какой-либо регистр, мы всегда подразумеваем, что там была создана идентичная копия и что оригинал все еще находится в ячейке 2000. Команды перемещения данных лучше было бы назвать командами дублирования данных, но термин «перемещение данных» уже устоялся.

Есть две причины, по которым данные могут копироваться из одного места в другое. Одна из них фундаментальна: присваивание переменным значений. Операция присваивания

```
A=B
```

выполняется путем копирования значения, которое находится в ячейке памяти с адресом B, в ячейку A, поскольку программист приказал это сделать. Вторая причина копирования данных — предоставить возможность быстрого обращения к ним. Как мы уже видели, многие команды могут обращаться к переменным только в том случае, если они имеются в регистре. Поскольку существует два возможных источника элемента данных (память и регистр) и существует два возможных пункта назначения для элемента данных (память и регистр), следовательно, существует 4 различных способа копирования. В одних компьютерах содержится 4 команды для 4 случаев, в других — одна команда для всех 4 случаев. Некоторые компьютеры используют команду `LOAD` для перемещения из памяти в регистр, команду `STORE` — для перемещения из регистра в память, команду `MOVE` — для перемещения из одного регистра в другой регистр, но не имеют никакой команды для копирования из одной части памяти в другую.

Команды перемещения данных должны как-то указывать, какое именно количество данных нужно переместить. Существуют команды для перемещения разного количества данных — от одного бита до всей памяти. В машинах с фиксированной длиной слова обычно перемещается ровно одно слово. Любые перемещения другого количества данных (больше слова или меньше слова) должны выполнять-

ся программным обеспечением с использованием сдвигов и слияний. Некоторые архитектуры команд дают возможность копировать отрезки данных размером меньше слова (они обычно измеряются в байтах), а также сразу несколько слов. Копирование нескольких слов рискованно, особенно если максимальное количество слов достаточно большое, поскольку такая операция может занять много времени и, возможно, ее придется прерывать в середине. Некоторые машины с изменяемой длиной слов содержат команды, которые определяют только адреса источника и места назначения, но не количество данных. Перемещение продолжается до тех пор, пока не появится специальное поле конца данных.

Бинарные операции

Бинарные операции — это такие операции, которые берут два операнда и получают из них результат. Все архитектуры команд содержат команды для сложения и вычитания целых чисел. Команды умножения и деления целых чисел также имеются практически во всех случаях. Думаю, нет необходимости объяснять, почему компьютеры оснащены арифметическими командами.

Следующая группа бинарных операций содержит булевы команды. Существует 16 булевых функций от двух переменных, но есть очень немного машин, в которых имеются команды для всех 16. Обычно присутствуют И, ИЛИ и НЕ; иногда кроме них еще есть ИСКЛЮЧАЮЩЕЕ ИЛИ, НЕ-ИЛИ и НЕ-И.

Важным применением команды И является выделение битов из слов. Рассмотрим машину со словами длиной 32 бита, в которой на одно слово приходится четыре 8-битных символа. Предположим, что нужно отделить второй символ от остальных трех, чтобы его напечатать. Это значит, что нужно создать слово, которое содержит этот символ в правых 8 битах с нулями в левых 24 битах (так называемое **выравнивание по правому биту**).

Чтобы извлечь нужный нам символ, слово, содержащее этот символ, соединяется операцией И с константой, которая называется **маской**. В результате этой операции все ненужные биты меняются на нули:

```
10110111 10111100 11011011 10001011 A
00000000 11111111 00000000 00000000 B (маска)
00000000 10111100 00000000 00000000 A И B
```

Затем результат сдвигается на 16 битов вправо, чтобы нужный символ находился в правом конце слова.

Важным применением команды ИЛИ является помещение битов в слово. Эта операция обратна операции извлечения. Чтобы изменить правые 8 битов 32-битного слова, не повредив при этом остальные 24 бита, сначала нежелательные 8 битов надо заменить на нули, а затем новый символ соединить операцией ИЛИ с полученным результатом, как показано ниже:

```
10110111 10111100 11011011 10001011 A
11111111 11111111 11111111 00000000 B (маска)
10110111 10111100 11011011 00000000 A И B
00000000 00000000 00000000 01010111 C
10110111 10111100 11011011 01010111 (A И B) ИЛИ C
```

Операция И убирает единицы, и в полученном результате никогда не бывает больше единиц, чем в любом из двух операндов. Операция ИЛИ вставляет единицы, и поэтому в полученном результате всегда по крайней мере столько же единиц, сколько в операнде с большим количеством единиц. Команда ИСКЛЮЧАЮЩЕЕ ИЛИ, в отличие от них, симметрична в отношении единиц и нулей. Такая симметрия иногда может быть полезной, например при порождении случайных чисел.

Большинство компьютеров сегодня поддерживают команды с плавающей точкой, которые в основном соответствуют арифметическим операциям с целыми числами. Большинство машин содержит по крайней мере 2 варианта таких чисел: более короткие для скорости и более длинные на тот случай, если требуется высокая точность вычислений. Существует множество возможных форматов для чисел с плавающей точкой, но сейчас практически везде применяется единый стандарт IEEE 754. Числа с плавающей точкой и этот стандарт обсуждаются в приложении Б.

Унарные операции

Унарные операции используют один операнд и производят один результат. Поскольку в данном случае нужно определять на один адрес меньше, чем в бинарных операциях, команды иногда бывают короче, хотя часто требуется определять другую информацию.

Команды для сдвига и циклического сдвига очень полезны. Они часто даются в нескольких вариантах. Сдвиги — это операции, при которых биты сдвигаются налево или направо, при этом биты, которые сдвигаются за пределы слова, утрачиваются. Циклические сдвиги — это сдвиги, при которых биты, вытесненные с одного конца, появляются на другом конце. Разница между обычным сдвигом и циклическим сдвигом показана ниже:

```
00000000 00000000 00000000 01110011 A
00000000 00000000 00000000 00011100 сдвиг вправо на 2 бита
11000000 00000000 00000000 00011100 циклический сдвиг вправо на 2 бита
```

Обычные и циклические сдвиги влево и вправо очень важны. Если n-битное слово циклически сдвигается влево на k битов, результат будет такой же, как при циклическом сдвиге вправо на n–k битов.

Сдвиги вправо часто выполняются с расширением по знаку. Это значит, что позиции, освободившиеся на левом конце слова, заполняются изначальным знаковым битом (0 или 1), как будто знаковый бит перетащили направо. Кроме того, это значит, что отрицательное число останется отрицательным. Ниже показаны сдвиги на 2 бита вправо:

```
1111111 11111111 11111111 11110000 A
0011111 11111111 11111111 11111100 A сдвинуто без знакового расширения
1111111 11111111 11111111 11111100 A сдвинуто со знаковым расширением
```

Операция сдвига используется при умножении и делении на 2. Если положительное целое число сдвигается влево на k битов, результатом будет изначальное число, умноженное на 2^k. Если положительное целое число сдвигается вправо на k битов, результатом будет изначальное число, деленное на 2^k.

Сдвиги могут использоваться для повышения скорости выполнения некоторых арифметических операций. Рассмотрим выражение 18×n, где n — положительное целое число. 18×n=16×n+2×n. 16×n можно получить путем сдвига копии n на 4 бита влево. 2×n можно получить, сдвинув n на 1 бит влево. Сумма этих двух чисел равна 18×n. Таким образом, это произведение можно вычислить путем одного перемещения, двух сдвигов и одного сложения, что обычно гораздо быстрее, чем сама операция умножения. Конечно, компилятор может применять такую схему, только если один из множителей является константой.

Сдвиг отрицательных чисел даже со знаковым расширением дает совершенно другие результаты. Рассмотрим, например, число −1 в обратном двоичном коде. При сдвиге влево на 1 бит получается число −3. При сдвиге влево еще на 1 бит получается число −7:

11111111 11111111 11111111 11111110 число −1 в обратном двоичном коде
11111111 11111111 11111111 11111100 число −1 сдвигается влево на 1 бит (−3)
11111111 11111111 11111111 11111000 число −1 сдвигается влево на 2 бита (−7)

Сдвиг влево отрицательных чисел в обратном двоичном коде не умножает число на 2. Однако сдвиг вправо производит деление корректно.

А теперь рассмотрим число −1 в дополнительном двоичном коде. При сдвиге вправо на 6 бит с расширением по знаку получается число −1, что неверно, поскольку целая часть от −1/64 равна 0:

11111111 11111111 11111111 11111111 число −1 в дополнительном двоичном коде
11111111 11111111 11111111 11111111 число −1, сдвинутое влево на 6 битов, равно −1

Как мы видим, сдвиг вправо вызывает ошибки. Однако при сдвиге влево число умножается на 2.

Операции циклического сдвига нужны для манипулирования последовательностями битов в словах. Если нужно проверить все биты в слове, при циклическом сдвиге слова последовательно по 1 биту каждый бит помещается в знаковый бит, где его можно легко проверить, а когда все биты проверены, можно восстановить изначальное значение слова. Операции циклического сдвига гораздо удобнее операций обычного сдвига, поскольку при этом не теряется информация: произвольная операция циклического сдвига может быть отменена другой операцией циклического сдвига.

В некоторых бинарных операциях очень часто используются совершенно определенные операнды, поэтому в архитектуры команд часто включаются унарные операции для их быстрого выполнения. Например, перемещение нуля в память или регистр чрезвычайно часто выполняется при начале вычислений. Перемещение нуля — это особый случай команды перемещения данных. Поэтому для повышения производительности часто вводится операция `CLR` с единственным адресом той ячейки, которую нужно очистить (то есть установить на 0).

Прибавление 1 к слову тоже часто используется при различных подсчетах. Унарная форма команды `ADD` — это операция `INC`, которая прибавляет 1. Другой пример — операция `NEG`. Отрицание X — это на самом деле бинарная операция вычитания 0−X, но поскольку операция отрицания очень часто применяется, в архитектуру

команд вводится команда NEG. Важно понимать разницу между арифметической операцией NEG и логической операцией NOT. Операция NEG производит **аддитивную инверсию** числа (такое число, сумма которого с изначальным числом дает 0). Операция NOT просто инвертирует все биты в слове. Эти операции очень похожи, а для системы, в которой используется представление в обратном двоичном коде, они идентичны. (В арифметике дополнительных кодов для выполнения команды NEG сначала инвертируются все биты, а затем к полученному результату прибавляется 1.)

Унарные и бинарные операции часто объединяются в группы по функциям, которые они выполняют, а вовсе не по числу операндов. В первую группу входят арифметические операции, в том числе операция отрицания. Во вторую группу входят логические операции и операции сдвига, поскольку эти две категории очень часто используются вместе для извлечения данных.

Сравнения и условные переходы

Практически все программы должны проверять свои данные и на основе результатов изменять последовательность команд, которые нужно выполнить. Рассмотрим функцию квадратного корня 0x. Если число x отрицательное, процедура сообщает об ошибке; если число положительное, процедура вычисляет квадратный корень. Функция *sqrt* должна проверять x, а затем совершать переход в зависимости от того, положительно число x или отрицательно.

Это можно сделать с помощью специальных команд условного перехода, которые проверяют какое-либо условие и совершают переход в определенный адрес памяти, если условие выполнено. Иногда определенный бит в команде указывает, нужно ли осуществлять переход в случае выполнения условия или в случае невыполнения условия соответственно. Часто целевой адрес является не абсолютным, а относительным (он связан с текущей командой).

Самое распространенное условие, которое нужно проверить, — равен ли определенный бит нулю или нет. Если команда проверяет знаковый бит числа и совершает переход к метке (LABEL) при условии, что бит равен 1, то если число было отрицательным, будут выполняться те утверждения, которые начинаются с метки LABEL, а если число было положительным или было равно 0, то будут выполняться те утверждения, которые следуют за условным переходом.

Во многих машинах содержатся биты кода условия, которые указывают на особые условия. Например, там может быть бит переполнения, который принимает значение 1 всякий раз, когда арифметическая операция выдает неправильный результат. Проверяя этот бит, мы проверяем выполнение предыдущей арифметической операции, и если произошла ошибка, то запускается программа обработки ошибок.

В некоторых процессорах есть специальный разряд (бит) переноса, который принимает значение 1, если происходит перенос из самого левого бита (например, при сложении двух отрицательных чисел). Бит переноса нельзя путать с битом переполнения. Проверка бита переноса необходима для вычислений с повышенной точностью (то есть когда целое число представлено двумя или более словами).

Проверка на ноль очень важна при выполнении циклов и в некоторых других случаях. Если бы все команды условного перехода проверяли только 1 бит, то тогда для проверки определенного слова на 0 нужно было бы отдельно проверять каждый бит, чтобы убедиться, что ни один бит не равен 1. Чтобы избежать подобной ситуации, во многие машины включается команда, которая должна проверять слово и осуществлять переход, если оно равно 0. Конечно же, это решение просто перекладывает ответственность на микроархитектуру. На практике аппаратное обеспечение обычно содержит регистр, все биты которого соединяются операцией ИЛИ, чтобы выдать на выходе один бит, по которому можно определить, содержит ли регистр биты, равные 1. Бит Z на рис. 4.1 обычно вычисляется следующим образом: сначала все выходные биты АЛУ соединяются операцией ИЛИ, а затем полученный результат инвертируется.

Операция сравнения слов или символов очень важна, например, при сортировке. Чтобы произвести сравнение, требуется три адреса: два нужны для элементов данных, а в третий адрес будет совершаться переход в случае выполнения условия. В тех компьютерах, где форматы команд позволяют содержать три адреса в команде, проблем не возникает. Но если такие форматы не предусмотрены, нужно что-то сделать, чтобы обойти эту проблему.

Одно из возможных решений — ввести команду, которая выполняет сравнение и записывает результат в один или несколько битов условия. Следующая команда может проверить биты условия и совершить переход, если два сравниваемых значения были равны, или неравны, или первое из них было больше второго и т. д. Такой подход применяется в Pentium II и UltraSPARC II.

В сравнении двух чисел есть некоторые тонкости. Сравнение — это не такая простая операция, как вычитание. Если очень большое положительно число сравнивается с очень большим отрицательным числом, операция вычитания приведет к переполнению, поскольку результат вычитания не может быть представлен. Тем не менее команда сравнения должна определить, удовлетворено ли условие, и возвратить правильный ответ. При сравнении не должно быть переполнений.

Кроме того, при сравнении чисел нужно решить, считаются ли числа числами со знаком или числами без знака. Трехбитные бинарные числа можно упорядочить двумя способами. От самого маленького к самому большому:

Без знака	Со знаком
000	100 (самое маленькое)
001	101
010	110
011	111
100	000
101	001
110	010
111	011 (самое большое)

В колонке слева приведены положительные числа от 0 до 7 по возрастанию. В колонке справа показаны целые числа со знаком от –4 до +3 в дополнительном двоичном коде. Ответ на вопрос: «Какое число больше: 011 или 100?» зависит от того, считаются ли числа числами со знаком. В большинстве архитектур есть команды для обращения с обоими типами упорядочения.

Команды вызова процедур

Процедура — это группа команд, которая выполняет определенную задачу и которую можно вызвать из нескольких мест программы. Вместо термина процедура часто используется термин **подпрограмма**, особенно когда речь идет о программах на языке ассемблера. Когда процедура закончила задачу, она должна вернуться к соответствующему оператору. Следовательно, адрес возврата должен как-то передаваться процедуре или сохраняться где-либо таким образом, чтобы можно было определить местонахождение после завершения задачи.

Адрес возврата может помещаться в одном из трех мест: в памяти, в регистре или в стеке. Самое худшее решение — поместить этот адрес в одну фиксированную ячейку памяти. Тогда если процедура будет вызывать другую процедуру, второй вызов приведет к потере первого адреса возврата.

Более удачное решение — сохранить адрес возврата в первом слове процедуры. Тогда первой выполняемой командой будет второе слово процедуры. После завершения процедуры происходит переход к первому слову, а если аппаратное обеспечение в первом слове наряду с адресом возврата дает код операции, то происходит непосредственный переход к этой операции. Процедура может вызывать другие процедуры, поскольку в каждой процедуре имеется пространство для одного адреса возврата. Но если процедура вызывает сама себя, эта схема не работает, поскольку первый адрес возврата будет уничтожен вторым вызовом. Способность процедуры вызывать саму себя, называемая **рекурсией,** очень важна и для теоретиков, и для практиков. Более того, если процедура A вызывает процедуру B, процедура B вызывает процедуру C, а процедура C вызывает процедуру A (непосредственная или цепочечная рекурсия), эта схема сохранения адреса возврата также не работает.

Еще более удачное решение — помещать адрес возврата в регистр. Если процедура рекурсивна, ей придется помещать адрес возврата в другое место каждый раз, когда она вызывается.

Самое лучшее решение — поместить адрес возврата в стек. Когда процедура завершена, она выталкивает адрес возврата из стека. При такой форме вызова процедур рекурсия не порождает никаких проблем; адрес возврата будет автоматически сохраняться таким образом, чтобы избежать уничтожения предыдущего адреса возврата. Мы рассматривали такой способ сохранения адреса возврата в машине IJVM (см. рис. 4.10).

Управление циклом

Часто возникает необходимость выполнять некоторую группу команд фиксированное количество раз, поэтому некоторые машины содержат команды для облегчения этого процесса. Все эти схемы содержат счетчик, который увеличивается или уменьшается на какую-либо константу каждый раз при выполнении цикла. Кроме того, этот счетчик каждый раз проверяется. При выполнении определенного условия цикл завершается.

Определенная процедура запускает счетчик вне цикла и затем сразу начинает выполнение цикла. Последняя команда цикла обновляет счетчик, и если условие завершения цикла еще не выполнено, то происходит возврат к первой команде цикла. Если условие выполнено, цикл завершается и начинается выполнение ко-

манды, идущей сразу после цикла. Цикл такого типа с проверкой в начале представлен в листинге 5.3. (Мы не могли здесь использовать язык Java, поскольку в нем нет оператора goto.)

Цикл такого типа всегда будет выполняться хотя бы один раз, даже если n≤0. Рассмотрим программу, которая поддерживает данные о персонале компании. В определенном месте программа начинает считывать информацию о конкретном работнике. Она считывает число n — количество детей у работника, и выполняет цикл n раз, по одному разу на каждого ребенка. Она считывает его имя, пол и дату рождения, так что компания может послать ему или ей подарок. Если у работника нет детей, n будет равно 0, но цикл все равно будет выполнен один раз, что даст ошибочные результаты.

В листинге 5.4 представлен другой способ выполнения проверки, который дает правильные результаты даже при n≤0. Отметим, что если одна команда выполняет и увеличение счетчика, и проверку условия, разработчики вынуждены выбирать один из двух методов.

Листинг 5.3. Цикл с проверкой в конце

```
        i=1;
L1:     первый оператор;
        .
        .
        .
        последний оператор;
        i=i+1;
        if (i<n) goto L1;
```

Листинг 5.4. Цикл с проверкой в начале

```
        i=1;
L1:     if(i>n) goto L2;
        первый  оператор;
        .
        .
        .
        последний оператор;
        i=i+1;
        goto L1;
L2:
```

Рассмотрим программу, которую нужно произвести для следующего выражения:

```
for (i=0; i<n; i++) {операторы}
```

Если у компилятора нет никакой информации о числе n, он должен применять подход, приведенный в листинге 5.4, чтобы корректно обработать случай n≤0. Однако если компилятор может определить, что n>0 (например, узнав, как определено n), он может использовать более удобный код, изложенный в листинге 5.3. Когда-то в стандарте языка FORTRAN требовалось, чтобы все циклы выполнялись один раз. Это позволяло всегда порождать более эффективный код (листинг 5.3). В 1977 году этот дефект был исправлен, поскольку даже приверженцы языка FORTRAN начали осознавать, что плохо иметь оператор цикла с такой странной семантикой, пусть он и позволяет экономить одну команду перехода на каждый цикл.

Команды ввода-вывода

Ни одна другая группа команд не различается настолько сильно в разных машинах, как команды ввода-вывода. В современных персональных компьютерах используются три различные схемы ввода-вывода:

1. Программируемый ввод-вывод с активным ожиданием.
2. Ввод-вывод с управлением по прерываниям.
3. Ввод-вывод с прямым доступом к памяти.

Мы рассмотрим каждую из этих схем по очереди.

Самым простым методом ввода-вывода является **программируемый ввод-вывод**, который часто используется в дешевых микропроцессорах, например во встроенных системах или в таких системах, которые должны быстро реагировать на внешние изменения (это системы, работающие в режиме реального времени). Эти процессоры обычно имеют одну входную и одну выходную команды. Каждая из этих команд выбирает одно из устройств ввода-вывода. Между фиксированным регистром в процессоре и выбранным устройством ввода-вывода передается один символ. Процессор должен выполнять определенную последовательность команд при каждом считывании и записи символа.

В качестве примера данного метода рассмотрим терминал с четырьмя 1-байтными регистрами, как показано на рис. 5.19. Два регистра используются для ввода: регистр состояния устройства и регистр данных. Два регистра используются для вывода: тоже регистр состояния устройства и регистр данных. Каждый из них имеет уникальный адрес. Если используется ввод-вывод с распределением памяти, все 4 регистра являются частью адресного пространства, и будут считываться и записываться с помощью обычных команд. В противном случае для чтения и записи регистров используются специальные команды ввода-вывода, например `IN` и `OUT`. В обоих случаях ввод-вывод осуществляется путем передачи данных и информации о состоянии устройства между центральным процессором и этими регистрами.

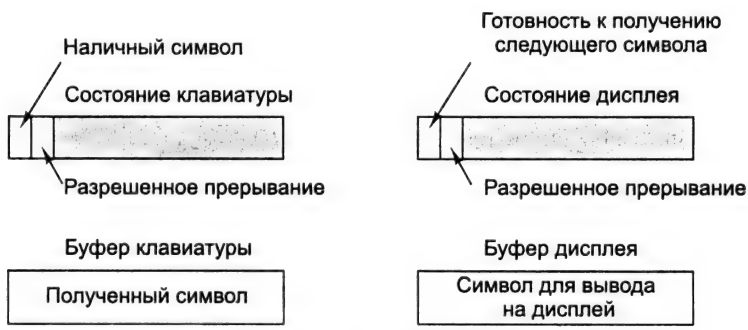


Рис. 5.19. Регистры устройств для простого терминала

Регистр состояния клавиатуры содержит 2 бита, которые используются, и 6 битов, которые не используются. Аппаратное обеспечение устанавливает самый левый бит на 1 всякий раз, когда появляется символ. Если программное обеспечение ранее установило на 1 бит 6, то производится прерывание. В противном случае прерывания не происходит. При программируемом вводе-выводе для получения входных данных центральный процессор обычно находится в цикле, периодически считывая регистр состояния клавиатуры, пока бит 7 не примет значение 1. Когда это случается, программное обеспечение считывает буферный регистр клавиатуры, чтобы получить символ. Считывание регистра данных вызывает установку бита CHARACTER AVAILABLE (наличия символа) на 0.

Вывод осуществляется сходным образом. Чтобы написать символ на экране, программное обеспечение сначала считывает регистр состояния дисплея, чтобы узнать, установлен ли бит READY (бит готовности) на 1. Если он не установлен,

программное обеспечение проходит цикл снова и снова до тех пор, пока данный бит не примет значение 1. Это значит, что устройство готово принять символ. Как только терминал приходит в состояние готовности, программное обеспечение записывает символ в буферный регистр дисплея, который переносит символ на экран и дает сигнал устройству установить бит готовности в регистре состояния дисплея на 0. Когда символ уже отображен, а терминал готов к обработке следующего символа, бит READY снова устанавливается на 1 контроллером.

В качестве примера программируемого ввода-вывода рассмотрим процедуру, написанную на Java (листинг 5.5). Эта процедура вызывается с двумя параметрами: массивом символов, который нужно вывести, и количеством символов, которые присутствуют в массиве (до 1 K). Тело процедуры представляет собой цикл, который выводит по одному символу. Сначала центральный процессор должен подождать, пока устройство будет готово, и только после этого он выводит символ, и эта последовательность действий повторяется для каждого символа. Процедуры *in* и *out* — это обычные процедуры языка ассемблера для чтения и записи регистров устройств, которые определяются по первому параметру, из или в переменную, которая определяется по второму параметру. Деление на 128 убирает младшие 7 битов, при этом бит READY остается в бите 0.

Листинг 5.5. Пример программируемого ввода-вывода

```
public static void output_buffer(int buf[], int count) {
       //Вывод блока данных на устройство
       int status, i, ready;

       for (i=0; i<count; i++) {
          do {
          status=in(display_status_reg);  // получение информации
                                             о состоянии устройства
          ready=(status<<7)&0x01;         // выделение бита
                                             готовности
    } while (ready==1);
    out(display_buffer_reg, buf[i]);
    }
    }
```

Основной недостаток программируемого ввода-вывода заключается в том, что центральный процессор проводит большую часть времени в цикле, ожидая готовности устройства. Такой процесс называется **активным ожиданием**. Если центральному процессору больше ничего не нужно делать (например, в стиральной машине), в этом нет ничего страшного (хотя даже простому контроллеру часто нужно контролировать несколько параллельных процессов). Но если процессору нужно выполнять еще какие-либо действия, например запускать другие программы, то активное ожидание здесь не подходит, и нужно искать другие методы ввода-вывода.

Чтобы избавиться от активного ожидания, нужно, чтобы центральный процессор запускал устройство ввода-вывода и указывал этому устройству, что необходимо осуществить запрос на прерывание, когда оно завершит свою работу. Посмотрите на рис. 5.19. Установив бит разрешения прерываний в регистре устройства, программное обеспечение сообщает, что аппаратное обеспечение будет передавать сигнал о завершении работы устройства ввода-вывода. Подробнее мы рассмотрим прерывания ниже в этой главе, когда перейдем к обсуждению передачи управления.

Во многих компьютерах сигнал прерывания порождается путем логического умножения (И) бита разрешения прерываний и бита готовности устройства. Если программное обеспечение сначала разрешает прерывание (перед запуском устройства ввода-вывода), прерывание произойдет сразу же, поскольку бит готовности будет равен 1. Таким образом, может понадобиться сначала запустить устройство, а затем сразу после этого ввести прерывание. Запись байта в регистр состояния устройства не изменяет бита готовности, который может только считываться.

Ввод-вывод с управлением по прерываниям — это большой шаг вперед по сравнению с программируемым вводом-выводом, но все же он далеко не совершенен. Дело в том, что прерывание требуется для каждого передаваемого символа. Следовательно, нужно каким-то образом избавиться от большинства прерываний.

Решение лежит в возвращении к программируемому вводу-выводу. Но только эту работу должен выполнять кто-то другой. Посмотрите на рис. 5.20. Мы добавили новую микросхему — контроллер **прямого доступа к памяти (ПДП)** с прямым доступом к шине.

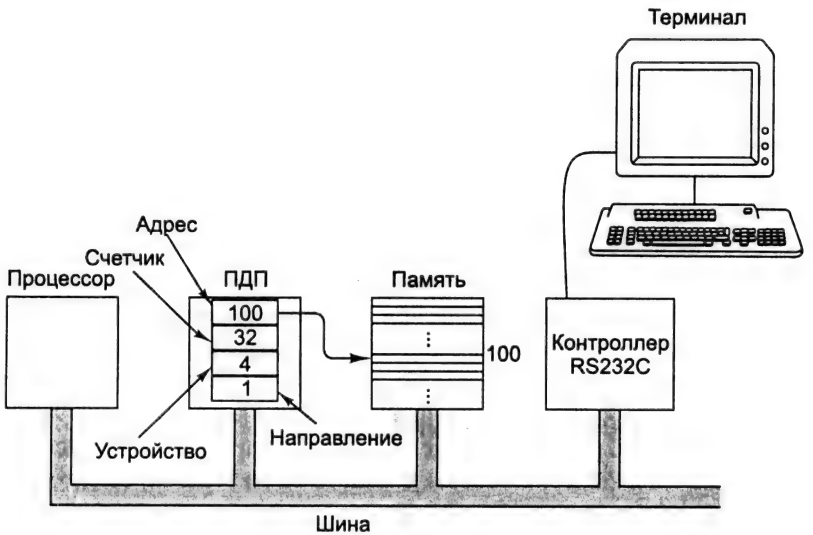


Рис. 5.20. Система с контроллером прямого доступа к памяти

Микросхема ПДП имеет по крайней мере 4 регистра. Все они могут загружаться программным обеспечением, работающим на центральном процессоре. Первый регистр содержит адрес памяти, который нужно считать или записать. Второй регистр содержит число, которое показывает количество передаваемых байтов или слов. Третий регистр содержит номер устройства или адрес устройства ввода-вывода, определяя, таким образом, какое именно устройство нам требуется. Четвертый регистр сообщает, должны ли данные считываться с устройства или записываться на него.

Чтобы записать блок из 32 байтов из адреса памяти 100 на терминал (например, устройство 4), центральный процессор записывает числа 32, 100 и 4 в первые три регистра ПДП и код записи (например, 1) в четвертый регистр, как показано

на рис. 5.20. Контроллер ПДП, инициализированный таким способом, делает запрос на доступ к шине, чтобы считать байт 100 из памяти, точно так же как если бы центральный процессор считывал этот байт. Получив нужный байт, контроллер ПДП посылает устройству 4 запрос на ввод-вывод, чтобы записать на него байт. После завершения этих двух операций контроллер ПДП увеличивает значение регистра адреса на 1 и уменьшает значение регистра счетчика на 1. Если значение счетчика больше 0, то следующий байт считывается из памяти и записывается на устройство ввода-вывода.

Когда значение счетчика доходит до 0, контроллер ПДП останавливает передачу данных и устанавливает линию прерывания на микросхеме процессора. При наличии ПДП центральному процессору нужно только инициализировать несколько регистров. После этого центральный процессор может выполнять какую-либо другую работу до тех пор, пока передача данных не завершится. При завершении передачи данных центральный процессор получает сигнал прерывания от контроллера ПДП. Некоторые контроллеры ПДП содержат два, три и более наборов регистров, так что они могут управлять несколькими процессами передачи одновременно.

Отметим, что если какое-нибудь высокоскоростное устройство, например диск, будет запускаться контроллером ПДП, то потребуется очень много циклов шины и для обращений к памяти, и для обращений к устройству. Во время этих циклов центральному процессору придется ждать (ПДП всегда имеет приоритет над центральным процессором на доступ к шине, поскольку устройства ввода-вывода обычно не допускают задержек). Процесс отбирания контроллером ПДП циклов шины у центрального процессора называется **захватом цикла**. Но выигрыш в том, что не нужно обрабатывать одно прерывание при каждом передаваемом байте (слове), сильно перевешивает потери, происходящие из-за захвата циклов.

Команды процессора Pentium II

В этом и следующих двух разделах мы рассмотрим наборы команд трех машин: Pentium II, UltraSPARC II и picoJava II. Каждая из них содержит базовые команды, которые обычно порождаются компиляторами, а также набор команд, которые редко используются или используются только операционной системой. Мы будем рассматривать обычные команды. Начнем с Pentium II.

Команды Pentium II представляют собой смесь команд 32-битного формата и команд, которые восходят к процессору 8088. На рисунке 5.21 приведены наиболее распространенные команды с целыми числами, которые широко используются в настоящее время. Этот список далеко не полный, поскольку в него не вошли команды с плавающей точкой, команды управления, а также некоторые редкие команды с целыми числами (например, использование 8-битного байта для выполнения поиска по таблице). Тем не менее этот список дает представление о том, какие действия может выполнять Pentium II.

Многие команды Pentium II обращаются к одному или к двум операндам, которые находятся или в регистрах, или в памяти. Например, бинарная команда `ADD`

складывает два операнда, а унарная команда INC увеличивает значение одного операнда на 1. Некоторые команды имеют несколько похожих вариантов. Например, команды сдвига могут сдвигать слово либо вправо, либо влево, а также могут рассматривать знаковый бит особо или нет. Большинство команд имеют несколько различных кодировок в зависимости от природы операндов.

На рисунке 5.21 поля SRC — это источники информации (SOURCE). Они не изменяются. Поля DST — это пункты назначения (DESTINATION). Они обычно изменяются командой. Существуют правила, определяющие, что может быть источником, а что пунктом назначения, но здесь мы не будем о них говорить. Многие имеют три варианта: для 8-, 16- и 32-битных операндов соответственно. Они различаются по коду операции И/ИЛИ по одному биту в команде. На рис. 5.21 приведены в основном 32-битные команды.

Для удобства мы разделили команды на несколько групп. Первая группа содержит команды, которые перемещают данные между частями машины: регистрами, памятью и стеком. Вторая группа содержит арифметические операции со знаком и без знака. Для умножения и деления 64-битное произведение или делимое хранится в двух регистрах: EAX (младшие биты) и EDX (старшие биты).

Третья группа включает двоично-десятичную арифметику. Здесь каждый байт рассматривается как два 4-битных **полубайта**. Каждый полубайт содержит 1 десятичный разряд (от 0 до 9). Комбинации битов от 1010 до 1111 не используются. Таким образом, 16-битное целое число может содержать десятичное число от 0 до 9999. Хотя такая форма хранения неэффективна, она устраняет необходимость переделывать десятичные входные данные в двоичные, а затем обратно в десятичные для вывода. Эти команды используются для выполнения арифметических действий над двоично-десятичными числами. Они широко используются в программах на языке COBOL.

Логические команды и команды сдвига манипулируют битами в слове или байте. Существует несколько комбинаций.

Следующие две группы связаны с проверкой и сравнением и осуществлением перехода в зависимости от полученного результата. Результаты проверки и сравнения хранятся в различных битах регистра EFLAGS. Значок Jxx стоит вместо набора команд, которые совершают условный переход в зависимости от результатов предыдущего сравнения (то есть в зависимости от битов в регистре EFLAGS).

В Pentium II есть несколько команд для загрузки, сохранения, перемещения, сравнения и сканирования цепочек символов или слов. Перед этими командами может стоять специальный префиксный байт REP (repetition — повторение), который заставляет команду повторяться до тех пор, пока не будет выполнено определенное условие (например, пока регистр ECX, значение которого уменьшается на 1 после каждого повторения, не будет равен 0). Таким образом, различные действия (перемещение, сравнение и т. д.) могут производиться над произвольными блоками данных.

Последняя группа содержит команды, которые не вошли ни в одну из предыдущих групп. Это команды перекодирования, команды управления, команды ввода-вывода и команды остановки процессора.

а

Команды перемещения

MOV DST, SRC	Перемещает SRC в DST
PUSH SRC	Помещает SRC в стек
POP DST	Выталкивает слово из стека и помещает его в DST
XCHG DS1, DS2	Меняет местами DS1 и DS2
LEA DST, SRC	Загружает действительный адрес SRC в DST
CMOV DST, SRC	Условное перемещение

б

Арифметические команды

ADD DST, SRC	Складывает SRC и DST
SUB DST, SRC	Вычитает SRC из DST
MUL SRC	Умножает EAX на SRC (без знака)
IMUL SRC	Умножает EAX на SRC (со знаком)
DIV SRC	Делит EDX:EAX на SRC (без знака)
IDV SRC	Делит EDX:EAX на SRC (со знаком)
ADC DST, SRC	Складывает SRC с DST и прибавляет бит переноса
SBB DST, SRC	Вычитает DST и переносит из SRC
INC DST	Прибавляет 1 к DST
DEC DST	Вычитает 1 из DST
NEG DST	Отрицает DST (вычитает DST из 0)

в

Двоично-десятичные команды

DAA	Десятичная коррекция
DAS	Десятичная коррекция для вычитания
AAA	Коррекция кода ASCII для сложения
AAS	Коррекция кода ASCII для вычитания
AAM	Коррекция кода ASCII для умножения
AAD	Коррекция кода ASCII для деления

г

Логические команды

AND DST, SRC	Логическая операция И над SRC и DST
OR DST, SRC	Логическая операция ИЛИ над SRC и DST
XOR DST, SRC	Логическая операция ИСКЛЮЧАЮЩЕЕ ИЛИ над SRC и DST
NOT DST	Замещение DST дополнением до 1

д

Команды сдвига/циклического сдвига

SAL/SAR DST, #	Сдвиг DST влево/вправо на # битов
SHL/SHR DST, #	Логический сдвиг DST влево/вправо на # битов
ROL/ROR DST, #	Циклический сдвиг DST влево/вправо на # битов
ROL/ROR DST, #	Циклический сдвиг DST по переносу на # битов

е

Команды тестирования/сравнения

TST SRC1, SRC2	Операнды логической операции И, установка флагов
CMP SRC1, SRC2	Установка флагов на основе вычитания SRC1-SRC2

Рис. 5.21. Команды с целыми числами в Pentium II (начало)

Команды передачи управления

ж

JMP ADDR	Переход к адресу
Jxx ADDR	Условные переходы на основе флагов
CALL ADDR	Вызов процедуры по адресу
RET	Выход из процедуры
IRET	Выход из прерывания
LOOPxx	Продолжает цикл до удовлетворения определенного условия
INT ADDR	Инициирует программное прерывание
INTO	Совершает прерывание, если установлен бит переполнения

Команды для операций над цепочками

з

LODS	Загружает цепочку
STOS	Сохраняет цепочку
MOVS	Перемещает цепочку
CMPS	Сравнивает две цепочки
SCAS	Сканирование цепочки

Коды условия

и

STC	Устанавливает бит переноса в регистре EFLAGS
CLC	Сбрасывает бит переноса в регистре EFLAGS
CMC	Образует дополнение бита переноса в регистре EFLAGS
STD	Устанавливает бит направления в регистре EFLAGS
CLD	Сбрасывает бит направления в регистре EFLAGS
STI	Устанавливает бит прерывания в регистре EFLAGS
CLI	Сбрасывает бит прерывания в регистре EFLAGS
PUSHFD	Помещает регистр EFLAGS в стек
POPFD	Выталкивает содержимое регистра EFLAGS из стека
LAHF	Загружает AH из регистра EFLAGS
SAHF	Сохраняет AH в регистре EFLAGS

Прочие команды

к

SWAP DST	Изменяет порядок байтов DST
CWQ	Расширяет EAX до EDX:EAX для деления
SWDE	Расширяет 16-битное число в AX до EAX
ENTER SIZE, LV	Создает стековый фрейм с байтами размера
LEAVE	Удаляет стековый фрейм, созданный командой ENTER
NOP	Пустая операция
HLT	Останов
IN AL, PORT	Переносит байт из порта в АЛУ
OUT PORT, AL	Переносит байт из АЛУ в порт
WAIT	Ожидает прерывания

SRC = источник (source); # = на сколько битов происходит сдвиг;
DST = пункт назначения (destination); LV = # локальных переменных

Рис. 5.21. Команды с целыми числами в Pentium II (окончание)

Pentium II имеет ряд **префиксов**. Один из них (REP) мы уже упомянули. Префикс — это специальный байт, который может ставиться практически перед любой командой (подобно WIDE в IJVM). Префикс REP заставляет команду, идущую за ним, повторяться до тех пор, пока регистр ECX не примет значение 0, как было сказано выше. REPZ и REPNZ заставляют команду выполняться снова и снова, пока код выполнения условия Z не примет значение 1 или 0 соответственно. Префикс LOCK резервирует шину для всей команды, чтобы можно было осуществлять многопроцессорную синхронизацию. Другие префиксы используются для того, чтобы команда работала в 16-битном или 32-битном формате. При этом не только меняется длина операндов, но и полностью переопределяются способы адресации.

Команды UltraSPARC II

Все целочисленные команды пользовательского режима UltraSPARC II приведены на рис. 5.22. Здесь не даются команды с плавающей точкой, команды управления (например, команды управления кэш-памятью, команды перезагрузки системы), команды, включающие адресные пространства, отличные от пользовательских, или устаревшие команды. Набор команд удивительно мал: UltraSPARC II — это процессор типа RISC.

Структура команд LOAD и STORE очень проста. Эти команды имеют варианты для 1, 2, 4 и 8 байтов. Если в 64-разрядный регистр загружается число размером меньше 64 битов, это число может быть либо расширено по знаку, либо дополнено нулями. Существуют команды для обоих вариантов.

Следующая группа команд предназначена для арифметических операций. Команды с буквами CC в названии устанавливают биты кода условия. На машинах CISC большинство команд устанавливают коды условия, но в машине типа RISC это нежелательно, поскольку ограничивает способность компилятора перемещать команды, стараясь заполнить отсрочки. Если изначальный порядок команд A...B...C, где A устанавливает коды условия, а B проверяет их, то компилятор не может вставить C между A и B, если C устанавливает условные коды. По этой причине многие команды имеют два варианта, при этом компилятор обычно использует ту команду, которая не устанавливает коды условия, если не планируется проверить их позже. Команды умножения, деления со знаком и деления без знака тоже поддерживаются.

Кроме этого, поддерживается специальный формат 30-битных чисел с автоматическим опознаванием типа данных за счет поля тега. Он используется для таких языков, как Smalltalk и Prolog, в которых тип переменных может меняться во время выполнения программы. При наличии таких чисел компилятор может породить команду ADD, а во время выполнения программы машина определяет, нужна ли в данном случае целочисленная команда ADD или команда ADD с плавающей точкой.

Группа команд сдвига включает одну команду сдвига влево и две команды сдвига вправо. Каждая из них имеет два варианта: 32-битный и 64-битный. Команды сдвига в основном используются для манипуляции с битами. Большинство машин CISC имеют довольно много различных команд обычного и циклического сдвига, и практически все они совершенно бесполезны.

Команды загрузки

а	LDSB ADDR, DST	Загружает байт со знаком (8 битов)
	LDUB ADDR, DST	Загружает байт без знака (8 битов)
	LDSH ADDR, DST	Загружает полуслово со знаком (8 битов)
	LDUH ADDR, DST	Загружает полуслово без знака (16 битов)
	LDSW ADDR, DST	Загружает слово со знаком (32 бита)
	LDUW ADDR, DST	Загружает слово без знака (32 бита)
	LDX ADDR, DST	Загружает расширенные слова (64 бита)

Команды сохранения

б	STB SRC, ADDR	Сохраняет байт (8 битов)
	STH SRC, ADDR	Сохраняет полуслово (16 битов)
	STW SRC, ADDR	Сохраняет слово (32 битов)
	STX SRC, ADDR	Загружает расширенное слово (64 бита)

Арифметические команды

в	ADD R1, S2, DST	Сложение
	ADDCC	Сложение с установкой кода условия
	ADDC	Сложение с переносом
	ADDCCC	Сложение с переносом и установкой кода условия
	SUB R1, S2, DST	Вычитание
	SUBCC	Вычитание с установкой кода условия
	SUBC	Вычитание с переносом
	SUBCCC	Вычитание с установкой кода переноса
	MULX R1, S2, DST	Умножение
	SDIVX R1, S2, DST	Деление со знаком
	UDIVX R1, S2, DST	Деление без знака
	TADCC R1, S2, DST	Сложение с использованием поля тега

Команды сдвига/циклического сдвига

г	SLL R1, S2, DST	Логический сдвиг влево (32 бита)
	SLLX R1, S2, DST	Логический сдвиг влево (64 бита)
	SRL R1, S2, DST	Логический сдвиг вправо (32 бита)
	SRLX R1, S2, DST	Логический сдвиг вправо (64 бита)
	SRA R1, S2, DST	Арифметический сдвиг вправо (32 бита)
	SRAX R1, S2, DST	Арифметический сдвиг вправо (64 бита)

Логические команды

д	AND R1, S2, DST "	Логическое И
	ANDCC "	Логическое И с установкой кода условия
	ANDN "	Логическое НЕ-И
	ANDNCC "	Логическое НЕ-И с установкой кода условия
	OR R1, S2, DST "	Логическое ИЛИ
	ORCC "	Логическое ИЛИ с установкой кода условия
	ORN "	Логическое НЕ-ИЛИ
	ORNCC "	Логическое НЕ-ИЛИ с установкой кода условия
	XOR R1, S2, DST "	Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ
	XORCC "	Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ с установкой кода условия
	XNOR "	Логическое ИСКЛЮЧАЮЩЕЕ НЕ-ИЛИ
	XNORCC "	Логическое ИСКЛЮЧАЮЩЕЕ НЕ-ИЛИ с установкой кода условия

Рис. 5.22. Основные целочисленные команды UltraSPARC II (начало)

Передача управления

	BPcc ADDR	Переход с прогнозированием
	BPr SRC, ADDR	Переход в регистр
	CALL ADDR	Вызов процедуры
	RETURN ADDR	Выход из процедуры
	JMPL ADDR, DST	Переход со связыванием
е	SAVE R1, S2, DST	Расширение регистровых окон
	RESTORE "	Восстановление регистровых окон
	Tcc CC, TRAP#	Системное прерывание при определенном условии
	PREFETCH FNC	Выборка данных из памяти с упреждением
	LDSTUB ADDR, R	Атомарная операция загрузки/сохранения
	MEMBAR MASK	Барьер памяти

Прочие команды

	SETHI CON, DST	Установка битов с 10 по 13
	MOVcc CC, S2, DST	Перемещение при определенном условии
	MOVr, R1, S2, DST	Перемещение в зависимости от значения регистра
	NOP	Пустая операция
ж	POPC S1, DST	Подсчет генеральной совокупности
	RDCCR V, DST	Чтение регистра кода условия
	WRCCR, R1, S2, V	Запись регистра кода условия
	RDPC V, DST	Чтение счетчика команд

SRC = входной регистр (source register);
DST = выходной регистр (destination redister);
R1 = входной регистр;
S2 = источник: регистр; или непосредственно получаемые данные;
ADDR = адрес памяти;

TRAP# = номер системного прерывания;
FCN = код функции;
MASK = тип операции;
CON = константа;
V = указатель регистра;

CC = набор кодов условия;
R = выходной регистр;

cc = условие;
r = LZ, LEZ, Z, NZ, GZ, GEZ

Рис. 5.22. Основные целочисленные команды UltraSPARC II (окончание)

Логические команды аналогичны арифметическим. Эта группа включает команды AND (И), OR(ИЛИ), EXCLUSIVE OR (ИСКЛЮЧАЮЩЕЕ ИЛИ), ANDN (НЕ-И), ORN (НЕ-ИЛИ) и XNOR (ИСКЛЮЧАЮЩЕЕ НЕ-ИЛИ). Значение последних трех команд спорно, но они могут выполняться за один цикл и не требуют практически никакого дополнительного аппаратного обеспечения, поэтому они часто включаются в набор команд. Даже разработчики машин RISC порой поддаются искушению.

Следующая группа содержит команды передачи управления. BPcc представляет собой набор команд, которые совершают переходы при различных условиях и определяют прогноз компилятора по поводу перехода. Команда BPr проверяет регистр и совершает переход, если условие подтвердилось.

Предусмотрено два способа вызова процедур. Для команды CALL используется формат 4 (см. рис. 5.10) с 30-битным смещением. Этого значения достаточно для того, чтобы добраться до любой команды в пределах 2 Гбайт от вызывающего оператора в любом направлении. Команда CALL копирует адрес возврата в регистр R15, который после вызова превращается в регистр R31.

Второй способ вызова процедуры — команда JMPL, для которой используется формат 1a или 1b, позволяющая помещать адрес возврата в любой регистр. Такая форма может быть полезной в том случае, если целевой адрес вычисляется во время выполнения.

Команды SAVE и RESTORE манипулируют регистровым окном и указателем стека. Обе команды совершают прерывание, если следующее (предыдущее) окно недоступно.

В последней группе содержатся команды, которые не попали ни в одну из групп. Команда SETHI необходима, поскольку невозможно поместить 32-битный непосредственный операнд в регистр. Для этого команда SETHI устанавливает биты с 10 по 31, а затем следующая команда передает оставшиеся биты, используя непосредственный формат.

Команда POPC подсчитывает число битов со значением 1 в слове. Последние три команды предназначены для чтения и записи специальных регистров.

Ряд широко распространенных команд CISC, которые отсутствуют в этом списке, можно легко получить, используя либо регистр G0, либо операнд-константу (формат 1b). Некоторые из них даны в табл. 5.9. Эти команды узнаются ассемблером UltraSPARC II и часто порождаются компиляторами. Многие из них используют тот факт, что регистр G0 связан с 0 и что запись в этот регистр не произведет никакого результата.

Команды компьютера picoJava II

Настало время рассмотреть уровень команд машины picoJava II. Здесь реализован полный набор команд JVM (226 команд), а также 115 дополнительных команд, предназначенных для C, C++ и операционной системы. Мы сосредоточимся главным образом на командах JVM, поскольку компилятор Java производит только эти команды. Архитектура команд JVM не содержит регистров, доступных пользователю, а также не имеет некоторых других особенностей, обычных для большинства центральных процессоров. (В процессоре picoJava II есть 64 встроенных регистра для вершины стека, но пользователи их не видят.) Большинство команд JVM помещают слова в стек, оперируют словами, находящимися в стеке, и выталкивают слова из стека. Большинство команд JVM выполняются непосредственно аппаратным обеспечением picoJava II, но некоторые из них микропрограммируются, а некоторые даже передаются программе обработки для выполнения.

Таким образом, для того чтобы заставить машину работать, требуется небольшая система уровня команд, но эта система гораздо меньше по размеру, чем полный интерпретатор JVM, и вызывается она только в редких случаях. Она содержит код для интерпретации нескольких сложных команд, загрузчик класса, верификатор байт-кода, администратор потока и программу чистки памяти («сборщик мусора»).

Таблица 5.9. Некоторые моделируемые команды UltraSPARC II

Команда	Как получить команду
MOV SRC, DST	Выполнить команду OR над SRC и G0 и сохранить результат в DST
CMP SRC1, SRC2	Вычесть SRC2 из SRC1(команда SUBCC) и сохранить результат в G0
TST SRC	Выполнить команду ORCC над SRC и G0 и сохранить результат в G0
NOT DST	Выполнить команду XNOR над DST и G0
NEG DST	Вычесть SRC2 из SRC1(команда SUBCC) и сохранить результат в G0
INC DST	Прибавить 1 к DST (непосредственный операнд) — команда ADD
DEC DST	Отнять 1 от DST (непосредственный операнд) — команда SUB
CLR DST	Выполнить команду OR над G0 и G0 и сохранить результат в DST
NOP	SETHI G0 на 0
RET	JMPL % I7+8, %G0

JVM содержит относительно небольшой набор простых команд. Набор всех команд JVM (за исключением некоторых расширенных, коротких и быстрых вариантов команд) приведен на рис. 5.23.

Команды JVM типизированы. Одну и ту же операцию с разными типами данных выполняют разные команды. Например, команда ILOAD помещает в стек целое 32-битное число, а команда ALOAD помещает в стек 32-битный указатель. Такое строгое разделение необязательно для правильного выполнения программы, поскольку в обоих случаях 32 бита, которые находятся в определенной ячейке памяти, передаются в стек независимо от типа этого 32-битного слова. Такое жесткое разграничение типов требуется для того, чтобы можно было проверить во время выполнения программы, не нарушены ли какие-нибудь ограничения (например, не пытается ли программа превратить целое число в указатель, чтобы обратиться к памяти).

Перейдем к командам JVM. Первая команда в списке — typeLOAD IND8.

На самом деле это не одна команда, а шаблон для порождения команд. Команды JVM регулярны, поэтому вместо того чтобы приводить все команды по одной, в некоторых случаях мы будем давать правило для порождения команд. В данном случае слово type заменяет одну из четырех букв: I, L, F и D, которые соответствуют типам integer (целые числа), long, float (32-битные числа с плавающей точкой) и double (64-битные числа с плавающей точкой) соответственно. Следовательно, здесь подразумевается 4 команды (ILOAD, LLOAD, FLOAD и DLOAD), каждая из которых содержит 8-битный индекс IND8 для нахождения локальной переменной и помещает в стек значение соответствующей длины и типа. Мы рассматривали только одну из этих команд — ILOAD, но остальные действуют точно так же и отличаются только по числу слов, помещаемых в стек, и по типу значения.

Кроме этих четырех команд существует еще четыре команды загрузки typeALOAD. Эти команды помещают в стек элементы массива. Во всех четырех случаях сначала в стек должен загружаться указатель на массив и индекс элемента массива. Затем эти команды выталкивают индекс и указатель, производят вычисление, чтобы найти элемент массива, и помещают этот элемент в стек. При вычислении нужно знать размер элементов, который определяется по типу. Эти команды получают индекс массива из стека, поэтому сама команда не содержит операнда.

Команды загрузки

а	typeLOAD IND8	Помещает локальную переменную в стек
	typeALOAD	Помещает элемент массива в стек
	BALOAD	Помещает байт из массива в стек
	SALOAD	Помещает short integer из массива в стек
	CALOAD	Помещает символ из массива в стек
	AALOAD	Помещает указатель из массива в стек

Команды сохранения

б	typeSTORE IND8	Выталкивает из стека значение и сохраняет его в локальной переменной
	typeASTORE	Выталкивает из стека значение и сохраняет его в массиве
	BASTORE	Выталкивает из стека байт и сохраняет его в массиве
	SASTORE	Выталкивает из стека short и сохраняет его в массиве
	CASTORE	Выталкивает из стека символ и сохраняет его в массиве
	AASTORE	Выталкивает из стека указатель и сохраняет его в массиве

Команды помещения в стек

в	BIPUSH CON8	Помещает небольшую константу в стек
	SIPUSH CON16	Помещает 16-битную константу в стек
	LDC IND8	Помещает в стек константу из набора констант
	typeCONST_*	Помещает в стек непосредственную константу
	ACONS_NULL	Помещает в стек нулевой указатель

Арифметические команды

г	typeADD	Сложение
	typeSUB	Вычитание
	typeMUL	Умножение
	typeDIV	Деление
	typeREM	Остаток
	typeNEG	Отрицание

Логические команды/команды сдвига

д	ilAND	Логическое И
	ilOR	Логическое ИЛИ
	ilXOR	Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ
	ilSHL	Сдвиг влево
	ilSHR	Сдвиг вправо
	ilUSHR	Сдвиг вправо без знака

Команды преобразования

е	x2y	Преобразует x в y
	i2c	Преобразует целое число в символ
	i2b	Преобразует целое число в байт

Команды управления стеком

ж	DUPxx	Шесть команд дублирования
	POP	Выталкивает целое число из стека и отбрасывает его
	POP2	Выталкивает два целых числа из стека и отбрасывает их
	SWAP	Меняет местами два верхних целых числа в стеке

Рис. 5.23. Набор команд JVM (начало)

Команды сравнения

IF_ICMPrel OFFSET16	Условный переход
IF_ACMPEQ OFFSET16	Переход в случае равенства двух значений
IF_ACMPNE OFFSET16	Переход в случае неравенства двух значений
IFrel OFFSET16	Проверяет одно значение и совершает переход
IFNULL OFFSET16	Совершает переход, если значение равно 0
IFNONNULL OFFSET16	Совершает переход, если значение не равно 0
LCMP	Сравнивает два числа long
FCMPL	Сравнивает два числа с плавающей точкой на <
FCMPG	Сравнивает два числа с плавающей точкой на >
DCMPL	Сравнивает два числа типа double на <
DCMPG	Сравнивает два числа типа double на >

Команды передачи управления

INVOKEVIRTUAL IND16	Вызов процедуры
INVOKESTATIC IND16	Вызов процедуры
INVOKEINTRFACE	Вызов процедуры
INVOKESPECIAL IND16	Вызов процедуры
JSR OFFSET16	Вызов процедуры
tipeRETURN	Возвращает значение
ARETURN	Возвращает указатель
RETURN	Возвращает пустой тип
RET IND8	Выход из процедуры
GOTO OFFSET16	Безусловный переход

Операции с массивами

ANEWARRAY IND16	Создает массив переменных
NEWARRAY ATYPE	Создает массив из массивов
MULTINEWARRAY IN16, D	Создает многомерный массив
ARRAYLENGTH	Выдает длину массива

Прочие команды

IINC IND8, CON16	Увеличивает локальную переменную на 1
WIDE	Префикс
NOP	Пустая операция
GETFIELD IND16	Считывает поле из объекта
PUTFIELD IND16	Записывает слово в объект
GETSTATIC IND16	Получает статическое поле из класса
NEW IND16	Создает новый объект
INSTANCEOF OFFSET16	Определяет тип объекта
CHECKCAST IND16	Проверяет тип объекта
ATHROW	Обработка исключения
LOOKUPSWITCH...	Разбросанные многоуровневые переходы
TABLESWITCH...	Компактные многоуровневые переходы
MONITORENTER	Входит в управляющую прграмму
MONITOREXIT	Выходит из управляющей прграммы

IND 8/16 = индекс локальной переменной;
CON 8/16, D, ATYPE = константа;

type, x, y = I, L, F, D;
OFFSET 16 для команд перехода

Рис. 5.23. Набор команд JVM (окончание)

Последние четыре команды этой группы также предназначены для работы с элементами массива, но только других типов. Они поддерживают byte (байт — 8 битов), short (16 битов), char (символ — 16 битов) и pointer (указатель — 32 бита). Таким образом, всего существует 12 команд LOAD.

Команды typeSTORE обратны командам typeLOAD. Каждая команда выталкивает элемент из стека и сохраняет его в локальной переменной. Одну из этих команд (ISTORE) мы уже рассматривали, когда изучали машину IJVM. Здесь также имеются команды для сохранения элементов массива. В вершине стека находится значение нужного типа. Под ним находится индекс, а еще ниже — указатель на массив. Все три элемента удаляются из стека этой командой.

Команды PUSH помещают значение в стек. Команду BIPUSH мы уже рассматривали. Команда SIPUSH выполняет ту же операцию, но только с 16-битным числом. Команда LDC помещает в стек значение из набора констант. Следующая команда представляет целую группу команд всех четырех основных типов (integer, long, float и double). Каждая команда содержит только код операции, и каждый код операции помещает определенное значение в стек. Например, команда ICONST_0 (код операции 0x03) помещает в стек 32-битное слово 0. То же самое действие можно произвести с помощью команды BIPUSH, но это займет два байта. Благодаря оптимизации самых распространенных команд программы JVM получаются небольшими по размеру. Поддерживаются следующие значения: Integers (целые числа) –1, 0, 1, 2, 3, 4, 5; longs 0 и 1; floats (числа с плавающей точкой) 0,0, 1,0 и 2,0 и doubles 0,0 и 1,0. Команда CONST_NULL помещает в стек нулевой указатель. Сочетание кодов операций и самых распространенных адресов в одной 1-байтной команде сильно сокращает размер команды, что экономит память и время на передачу бинарных программ на языке Java по Интернету.

Арифметические операции абсолютно регулярны. Для каждого из основных четырех типов имеется 4 команды. Три логические операции и три операции сдвига применяются только для целых чисел и чисел типа long. Наличие команды AND для чисел с плавающей точкой противоречило бы строгим правилам типизирования JVM. Строка x2y в таблице представляет 4×4 команд преобразования. Значения каждого типа могут переделываться в любой другой тип (но только не в тот же самый), поэтому здесь имеется 12 команд. Самой типичной из них является команда I2F, которая превращает целое число в число с плавающей точкой.

Группа команд управления стеком содержит команды, которые дублируют верхнее значение или два значения стека и помещают их в различные части стека (не обязательно в вершину). Остальные команды выталкивают значения из стека и меняют местами два верхних значения.

Команды группы сравнения выталкивают одно или два значения из стека и проверяют их. Если из стека выталкивается два значения, то одно из них вычитается из другого, а результат проверяется. Если выталкивается одно значение, то оно и проверяется. Суффикс rel замещает реляционные операторы: LT, LE, EQ, NE, GE и GT. Команды со смещением совершают переход, если определенное условие подтверждено. Остальные команды помещают результат обратно в стек.

Следующая группа предназначена для вызова процедур и возвращения значений. При изучении машины IJVM мы рассматривали очень простые версии команд INVOKEVIRTUAL и IRETURN. Полные версии содержат гораздо больше параметров,

и существует множество команд, которые покрывают самые различные случаи. В этой книге мы не будем описывать эти команды. Подробнее см. [85].

Еще одну группу образуют 4 команды для создания одномерных и многомерных массивов и проверки их длины. В машине JVM массивы хранятся в «куче» и периодически очищаются (процесс «сборки мусора»), когда они уже больше не нужны.

Последняя группа включает в себя оставшиеся команды. Каждая из этих команд имеет специальное назначение, связанное с какой-нибудь особенностью языка Java. Описание этих команд не входит в задачи этой книги.

А теперь нужно сказать пару слов о самом уровне команд picoJava II. Это машина с обратным порядком байтов (хотя существует несколько команд, которые можно переделать в формат с прямым порядком байтов). Слова состоят из 32 битов, хотя существуют команды для работы с единицами по 8, 16 и 64 бита. Стек в памяти располагается от верхних адресов к нижним в отличие от IJVM (спецификация JVM допускает оба варианта).

Машина picoJava II была разработана для программ на Java, C и C++. Но чтобы запустить программы на C и C++, нужен компилятор, который превращает C и C++ в команды picoJava II. После того как программа на C или C++ была скомпилирована на JVM, все способы оптимизации аппаратного обеспечения, которые мы описывали в главе 4, становятся применимы для C и C++.

Чтобы программы на C и C++ могли работать на машине picoJava II, к уровню архитектуры команд было добавлено 115 дополнительных команд. Большинство из них составляют два или более байтов в длину и начинаются с одного из двух зарезервированных кодов JVM (0xFE и 0xFF), которые показывают, что дальше следует расширенная команда. Ниже мы дадим краткий обзор особенностей picoJava II, не характерных для JVM.

В машине picoJava II содержится 25 32-битных регистров. Четыре из них по функциям эквивалентны регистрам PC, LV, SP и CPP машины IJVM. Регистр OPLIM помещает определенное предельное значение в SP. Если значение SP выходит за пределы OPLIM, то происходит прерывание. Эта особенность позволяет представлять стек в виде связного списка участков стека, а не как один непрерывный блок памяти. Регистр FRAME отмечает конец фрейма локальных переменных и указывает на слово, которое содержит счетчик команд вызывающей процедуры.

Среди других регистров можно назвать слово состояния программы — это регистр, который следит, насколько заполнен 64-регистровый стековый кэш, и четыре регистра, которые используются для управления потоком. Кроме того, существует 4 регистра для ловушек и прерываний и 4 регистра для вызова процедур и возвращения значений в командах на языках C и C++. Поскольку picoJava II не имеет виртуальной памяти, для ограничения определенной части памяти, к которой может иметь доступ текущая программа на C или C++, используются два специальных регистра. Расширенные команды можно разделить на 5 категорий. К первой категории относятся команды для чтения и записи верхних регистров. Ко второй категории относятся команды для работы с указателями. Они позволяют считывать из памяти и записывать в память произвольные слова. Большинство из этих команд выталкивают машинный адрес из стека, а затем помещают в стек содержи-

мое байта, слова и т. д., находящегося в ячейке с этим адресом. Такие команды нарушают типовую безопасность языка Java, но они нужны для C и C++.

В третью категорию входят команды для программ на C и C++, например вызов процедур и выход из процедур без применения команд JVM. К четвертой группе относятся команды, которые управляют аппаратным обеспечением, например кэш-памятью. В пятую категорию включены команды разного рода, например проверки при включении. Программы, использующие эти дополнительные команды, не переносимы на другие машины JVM.

Сравнение наборов команд

Рассмотренные наборы команд очень сильно отличаются друг от друга. Pentium II — это классическая двухадресная 32-битная машина CISC. Она пережила долгую историю, у нее особые и нерегулярные способы адресации, и она содержит множество команд, которые обращаются к памяти. UltraSPARC II — это современная трехадресная 64-битная машина RISC с архитектурой загрузки/сохранения, всего двумя способами адресации и компактным и эффективным набором команд. JVM — это машина со стековой организацией, практически без способов адресации, с регулярными командами и очень плотным кодированием команд.

В основу разработки компьютера Pentium II легли три основных фактора:

1. Обратная совместимость.
2. Обратная совместимость.
3. Обратная совместимость.

При нынешнем положении вещей никто не стал бы разрабатывать такую нерегулярную машину с таким маленьким количеством абсолютно разных регистров. По этой причине очень сложно писать компиляторы. Из-за недостатка регистров компиляторам постоянно приходится сохранять переменные в памяти, а затем вновь загружать их, что очень невыгодно даже при наличии трех уровней кэш-памяти. Только благодаря таланту инженеров компании Intel процессор Pentium II работает достаточно быстро, несмотря на все недостатки уровня команд. Но, как мы увидели в главе 4, реализация этого процессора чрезвычайно сложна и требует транзисторов в два раза больше, чем picoJava II, и почти в полтора раза больше, чем UltraSPARC II.

Современная разработка уровня команд представлена в процессоре UltraSPARC II. Он содержит полную 64-битную архитектуру команд (с шиной на 128 битов). Процессор содержит много регистров и имеет набор команд, в котором преобладают трехрегистровые операции, а также имеется небольшая группа команд `LOAD` и `STORE`. Все команды одного размера, хотя число форматов вышло из-под контроля. Большинство новых разработок очень похожи на UltraSPARC II, но содержат меньше форматов команд.

JVM — машина совершенно другого рода. Здесь уровень команд изначально разрабатывался так, чтобы небольшие программы можно было передавать по Интернету и интерпретировать на программном обеспечении другого компьютера. Это была разработка для одного языка. Все это привело к использованию стека и коротким командам разной длины с очень высокой плотностью (в среднем всего

1,8 байта на команду). Создание аппаратного обеспечения, которое выполняет одну команду JVM за раз и при выполнении одной команды обращается к памяти два или три раза, кажется нонсенсом. Но благодаря помещению на микросхему стека из 64 слов и переделыванию целых последовательностей команд в современные трехадресные команды RISC машина picoJava II умудряется неплохо работать с очень неэффективной архитектурой команд.

Ядро современного компьютера представляет собой сильно конвейеризированное трехрегистровое устройство загрузки/сохранения типа RISC. UltraSPARC II просто открыто сообщает об этой структуре пользователю. Pentium II скрывает эту систему RISC, перенимая старую архитектуру команд и разбивая команды CISC на микрооперации RISC. Машина picoJava II также таит в себе ядро RISC, комбинируя несколько команд для получения одной команды RISC.

Поток управления

Поток управления — это последовательность, в которой команды выполняются динамически, то есть во время работы программы. При отсутствии переходов и вызовов процедур команды вызываются из последовательных ячеек памяти. Вызов процедуры влечет за собой изменение потока управления, выполняемая в данный момент процедура останавливается, и начинается выполнение вызванной процедуры. Сопрограммы связаны с процедурами и вызывают сходные изменения в потоке управления. Они нужны для моделирования параллельных процессов. Ловушки (traps) и прерывания тоже меняют поток управления при возникновении определенных ситуаций. Все это мы обсудим в следующих разделах.

Последовательный поток управления и переходы

Большинство команд не меняют поток управления. После выполнения одной команды вызывается и выполняется та команда, которая идет вслед за ней в памяти. После выполнения каждой команды счетчик команд увеличивается на число, соответствующее длине команды. Счетчик команд представляет собой линейную функцию от времени, которая увеличивается на среднюю длину команды за средний промежуток времени. Иными словами, процессор выполняет команды в том же порядке, в котором они появляются в листинге программы, как показано на рис. 5.24, *а*.

Если программа содержит переходы, то это простое соотношение между порядком расположения команд в памяти и порядком их выполнения больше не соответствует действительности. При наличии переходов счетчик команд больше не является монотонно возрастающей функцией от времени, как показано на рис. 5.24, *б*. В результате последовательность выполнения команд из самой программы уже не видна. Если программисты не знают, в какой последовательности процессор будет выполнять команды, это может привести к ошибкам. Такое наблюдение побудило Дейкстру [31] написать статью под названием «Оператор GOTO нужно считать вредным», в котором он предлагал избегать в программах оператора goto. Эта статья дала толчок революции в программировании, одним из нововведений которой было устранение операторов goto более структурированными формами потока управления, например циклами while. Конечно, эти про-

граммы компилируются в программы второго уровня, которые могут содержать многочисленные переходы, поскольку реализация операторов `if`, `while` и структур языков высокого уровня требует совершения переходов.

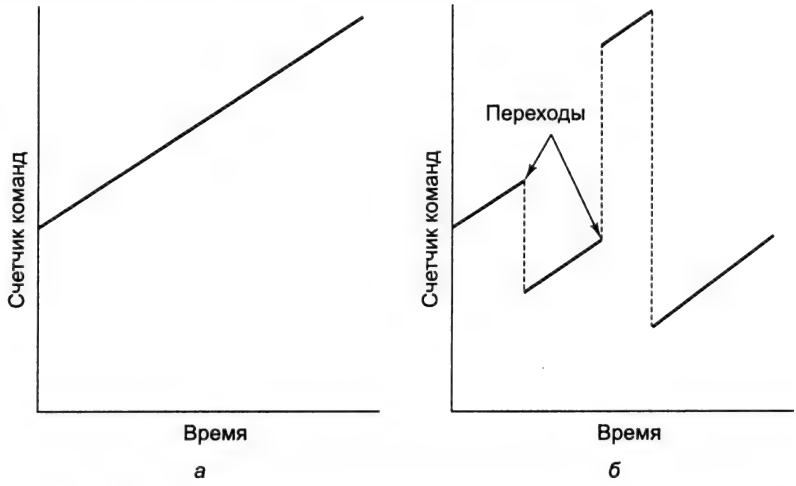


Рис. 5.24. Счетчик команд как функция от времени (приближенно): без переходов (*а*); с переходами (*б*)

Процедуры

Самым важным способом структурирования программ является процедура. С одной стороны, вызов процедуры, как и команда перехода, изменяет поток управления, но в отличие от команды перехода после выполнения задачи управление возвращается к команде, которая вызвала процедуру.

С другой стороны, тело процедуры можно рассматривать как определение новой команды на более высоком уровне. С этой точки зрения вызов процедуры можно считать отдельной командой, даже если процедура очень сложная. Чтобы понять часть программы, содержащую вызов процедуры, нужно знать, что она делает и как она это делает.

Особый интерес представляет **рекурсивная процедура**. Это такая процедура, которая вызывает сама себя либо непосредственно, либо через цепочку других процедур. Изучение рекурсивных процедур дает значительное понимание того, как реализуются вызовы процедур и что в действительности представляют собой локальные переменные. А теперь рассмотрим пример рекурсивной процедуры.

«Ханойская башня» — это древняя задача, которая имеет простое решение с использованием рекурсии. В одном монастыре в Ханое есть три золотых колышка. Вокруг первого из них располагались 64 концентрических золотых диска, каждый из них с отверстием посередине для колышка. Диаметр дисков уменьшается снизу вверх. Второй и третий колышки абсолютно пусты. Монахи переносят все диски на колышек 3 по одному диску, но диск большего размера не может находиться сверху на диске меньшего размера. Говорят, что когда они закончат, наступит конец света. Если вы хотите потренироваться, вы можете использовать пласти-

ковые диски, и не 64, а поменьше, но когда вы решите эту задачу, ничего страшного не произойдет. Чтобы произошел конец света, требуется 64 диска, и все они должны быть из золота. На рисунке 5.25 показана начальная конфигурация, где число дисков (n) равно 5.

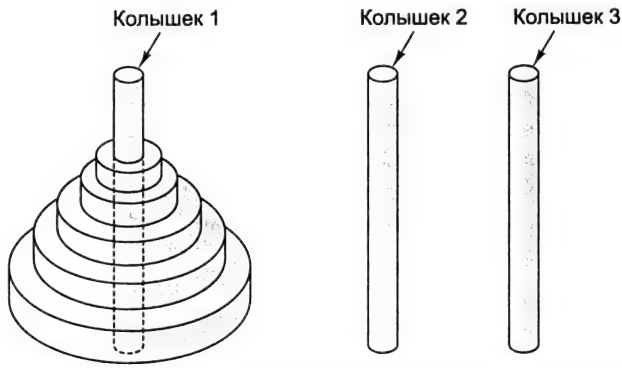


Рис. 5.25. Исходное положение в задаче «Ханойская башня» для пяти дисков

Чтобы переместить n дисков с колышка 1 на колышек 3, нужно сначала перенести n–1 дисков с колышка 1 на колышек 2, затем перенести один диск с колышка1 на колышек 3, а потом перенести n–1 диск с колышка 2 на колышек 3. Решение этой задачи проиллюстрировано на рис. 5.26.

Для решения задачи нам нужна процедура, которая перемещает n дисков с колышка i на колышек j. Когда эта процедура вызывается,

```
towers (n, i, j)
```

решение выводится на экран. Сначала процедура проверяет, равно ли n единице. Если да, то решение тривиально: нужно просто переместить один диск с i на j. Если n не равно 1, решение состоит из трех частей, как было сказано выше, и каждая из этих частей представляет собой рекурсивную процедуру.

Полное решение показано в листинге 5.6. Вызов процедуры

```
towers (3, 1, 3)
```

порождает еще три вызова

```
towers (2, 1, 2)
towers (1, 1, 3)
towers (2, 2, 3)
```

Первый и третий вызовы производят по три вызова каждый, и всего получится семь.

Листинг 5.6. Процедура для решения задачи «Ханойская башня»

```
public void towers (int n, int i, int j)
int k;
if (n==1)
System.out.println("Переместить диск из" + i + "на" + j); else
k=6-i-j;
towers(n-1, i, k);
towers (1, i, j);
towers (n-1, k, j);
```

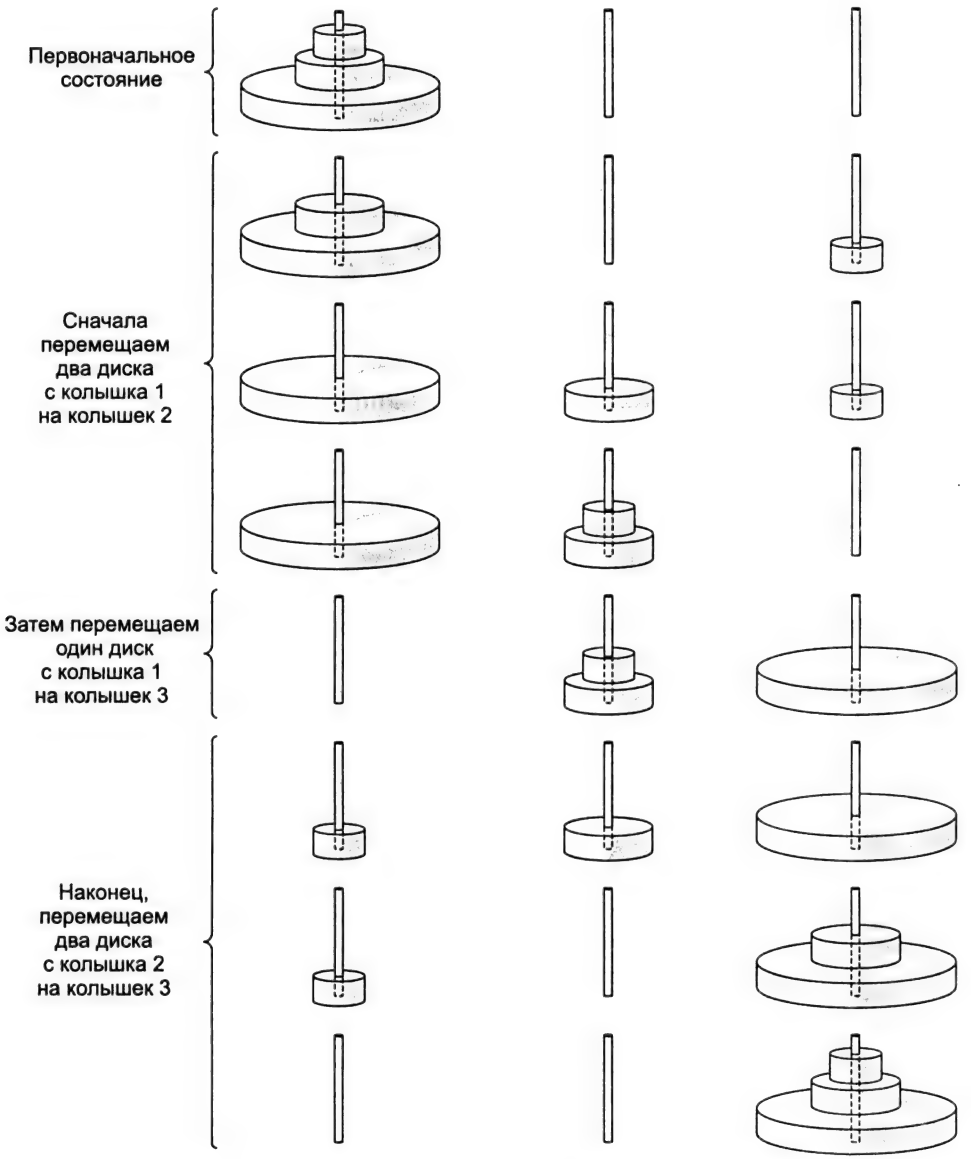


Рис. 5.26. Решение задачи «Ханойская башня» для трех дисков

Для рекурсивных процедур нам нужен стек, чтобы хранить параметры и локальные переменные для каждого вызова, как и в IJVM. Каждый раз при вызове процедуры на вершине стека новый стековый фрейм для процедуры. Текущий фрейм — это тот фрейм, который был создан последним. В наших примерах стек растет снизу вверх от малых адресов к большим, как и в IJVM.

Помимо указателя стека, который указывает на вершину стека, удобно иметь указатель фрейма (FP — Frame Pointer), который указывает на фиксированное

место во фрейме. Он может указывать на связующий указатель, как в IJVM, или на первую локальную переменную. На рис. 5.27 изображен стековый фрейм для машины с 32-битным словом. При первом вызове процедуры towers в стек помещаются n, i и j, а затем выполняется команда CALL, которая помещает в стек адрес возврата, 1012. Вызванная процедура сохраняет в стеке старое значение FP (1000) в ячейке 1016, а затем передвигает указатель стека для обозначения места хранения локальных переменных. При наличии только одной 32-битной локальной переменной (k) SP (Stack Pointer — указатель стека) увеличивается на 4 до 1020. На рис. 5.27, *а* показан результат всех этих действий.

Первое, что должна сделать процедура после того, как ее вызвали, — это сохранить предыдущее значение FP (так, чтобы его можно было восстановить при выходе из процедуры), скопировать значение SP в FP и, возможно, увеличить на одно слово, в зависимости от того, куда указывает FP нового фрейма. В этом примере FP указывает на первую локальную переменную, а в IJVM LV указывает на связующий указатель. Разные машины оперируют с указателем фрейма немного поразному, иногда помещая его в самый низ стекового фрейма, иногда — в вершину, а иногда — в середину, как на рис. 5.27. В этом отношении стоит сравнить рис. 5.27 с рис. 4.12, чтобы увидеть два разных способа обращения со связующим указателем. Возможны и другие способы. Но в любом случае обязательно должна быть возможность выйти из процедуры и восстановить предыдущее состояние стека.

Код, который сохраняет старый указатель фрейма, устанавливает новый указатель фрейма и увеличивает указатель стека, чтобы зарезервировать пространство для локальных переменных, называется **прологом процедуры**. При выходе из процедуры стек должен быть очищен, и этот процесс называется **эпилогом процедуры**. Одна из важнейших характеристик компьютера — насколько быстро он может совершать пролог и эпилог. Если они очень длинные и выполняются медленно, делать вызовы процедур будет невыгодно. Команды ENTER и LEAVE в машине Pentium II были разработаны для того, чтобы пролог и эпилог процедуры работали эффективно. Конечно, они содержат определенную модель обращения с указателем фрейма, и если компилятор имеет другую модель, их нельзя использовать.

А теперь вернемся к задаче «Ханойская башня». Каждый вызов процедуры добавляет новый фрейм к стеку, а каждый выход из процедуры удаляет фрейм из стека. Ниже мы проиллюстрируем, как используется стек при реализации рекурсивных процедур. Начнем с вызова

```
towers (3, 1, 3)
```

На рис. 5.27, *а* показано состояние стека сразу после вызова процедуры. Сначала процедура проверяет, равно ли n единице, а установив, что n=3, заполняет k и совершает вызов

```
towers (2, 1, 2)
```

Состояние стека после завершения этого вызова показано на рис. 5.27, *б*. После этого процедура начинается с начала (вызванная процедура всегда начинается с начала). На этот раз условие n=1 снова не подтверждается, поэтому процедура снова заполняет k и совершает вызов

```
towers (1, 1, 3)
```

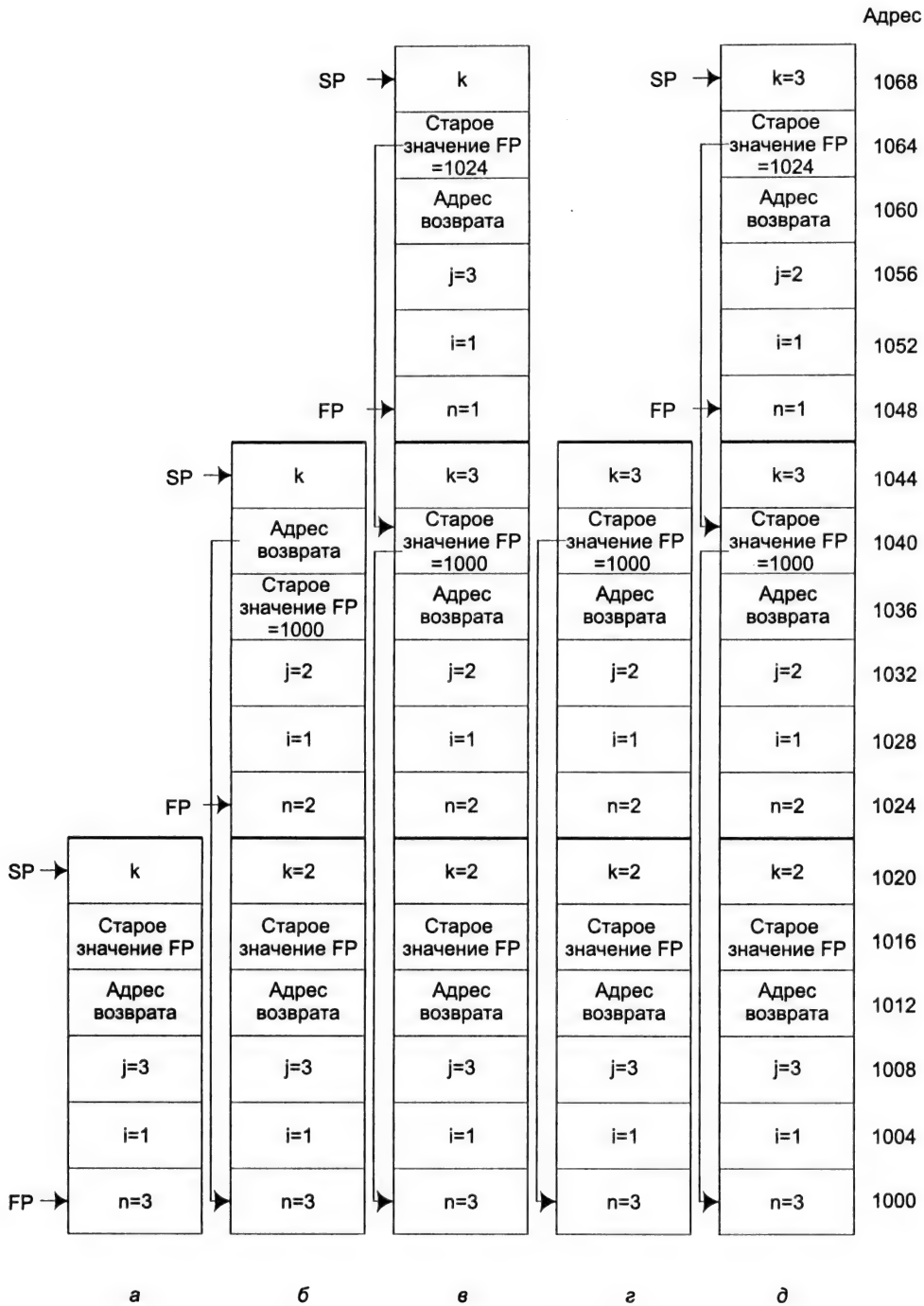


Рис. 5.27. Состояние стека во время выполнения программы листинга 5.6

Состояние стека после этого вызова показано на рис. 5.27, *в*. Счетчик команд указывает на начало процедуры. На этот раз условие подтверждается, и на экран выводится строка. Затем совершается выход из процедуры. Для этого удаляется один фрейм, а значения FP и SP переопределяются (см. рис. 5.27, *г*). Затем процедура продолжает выполняться в адресе возврата:

```
towers (1, 1, 2)
```

Это добавляет новый фрейм в стек (см. рис. 5.27, *д*). Печатается еще одна строка. После выхода из процедуры фрейм удаляется из стека. Вызовы процедур продолжаются до тех пор, пока не завершится выполнение первой процедуры и пока фрейм, изображенный на рис. 5.27, *а*, не будет удален из стека. Чтобы вы лучше смогли понять, как работает рекурсия, вам нужно произвести полное выполнение процедуры

```
towers (3, 1, 3)
```

используя ручку и бумагу.

Сопрограммы

В обычной последовательности вызовов существует четкое различие между вызывающей процедурой и вызываемой процедурой. Рассмотрим процедуру A, которая вызывает процедуру B (рис. 5.28).

Процедура B работает какое-то время, затем возвращается к A. На первый взгляд может показаться, что эти ситуации симметричны, поскольку ни A, ни B не являются главной программой. И A, и B — это процедуры. (Процедуру A можно было бы назвать основной программой, но это в данном случае неуместно.) Более того, сначала управление передается от A к B (при вызове), а затем — от B к A (при возвращении).

Различие состоит в том, что когда управление переходит от A к B, процедура B начинает выполняться с самого начала; а при переходе из B обратно в A выполнение начинается не с начала процедуры A, а с того момента, за которым последовал вызов процедуры B. Если A работает некоторое время, а потом снова вызывает процедуру B, выполнение B снова начинается с самого начала, а не с того места, после которого произошло возвращение к процедуре A. Если процедура A вызывает процедуру B много раз, процедура B каждый раз начинается с начала, а процедура A уже никогда больше с начала не начинается.

Это различие отражается в способе передачи управления между A и B. Когда A вызывает B, она использует команду вызова процедуры, которая помещает адрес возврата (то есть адрес того выражения, которое последует за процедурой) в такое место, откуда его потом легко будет вытащить, например в вершину стека. Затем она помещает адрес процедуры B в счетчик команд, чтобы завершить вызов. Для выхода из процедуры B используется не команда вызова процедуры, а команда выхода из процедуры, которая просто выталкивает адрес возврата из стека и помещает его в счетчик команд.

Иногда нужно иметь две процедуры A и B, каждая из которых вызывает другую в качестве процедуры, как показано на рис. 5.29. При возврате из B к A про-

цедура B совершает переход к тому оператору, за которым последовал вызов процедуры B. Когда процедура A передает управление процедуре B, она возвращается не к самому началу B (за исключением первого раза), а к тому месту, на котором произошел предыдущий вызов A. Две процедуры, работающие подобным образом, называются **сопрограммами**.

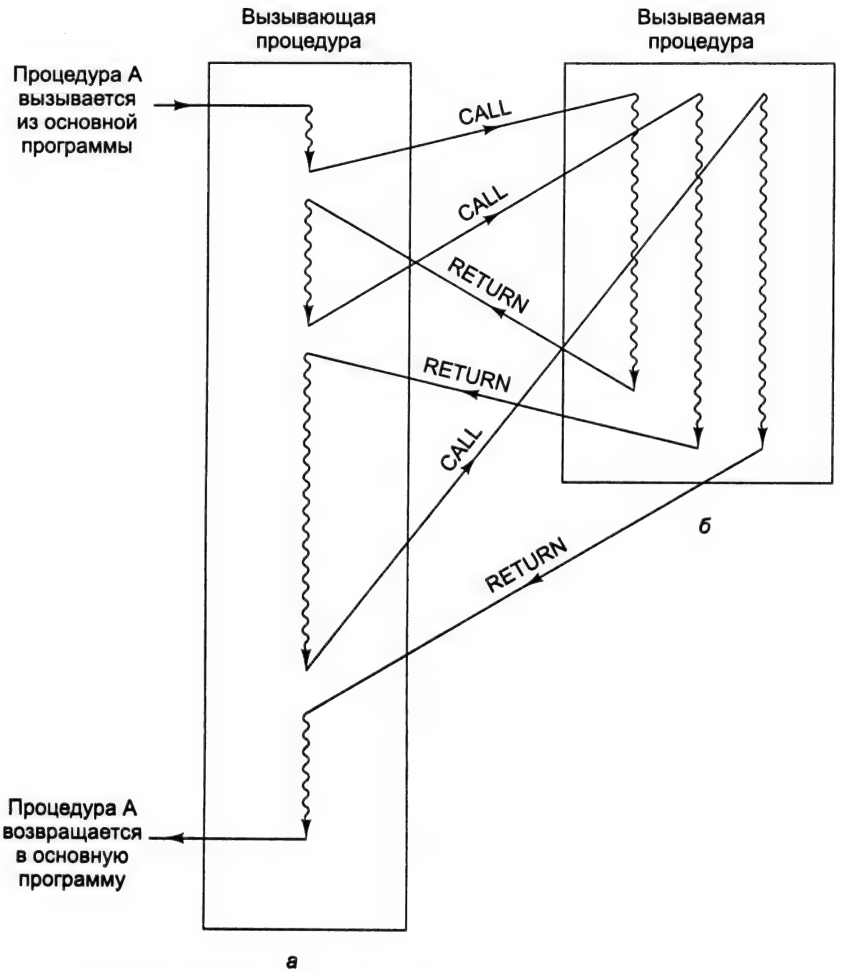


Рис. 5.28. Выполнение вызванной процедуры всегда начинается с самого начала этой процедуры

Сопрограммы обычно используются для того, чтобы производить параллельную обработку данных на одном процессоре. Каждая сопрограмма работает как бы параллельно с другими сопрограммами, как будто у нее есть собственный процессор. Такой подход упрощает программирование некоторых приложений. Он также полезен для проверки программного обеспечения, которое потом будет работать на мультипроцессоре.

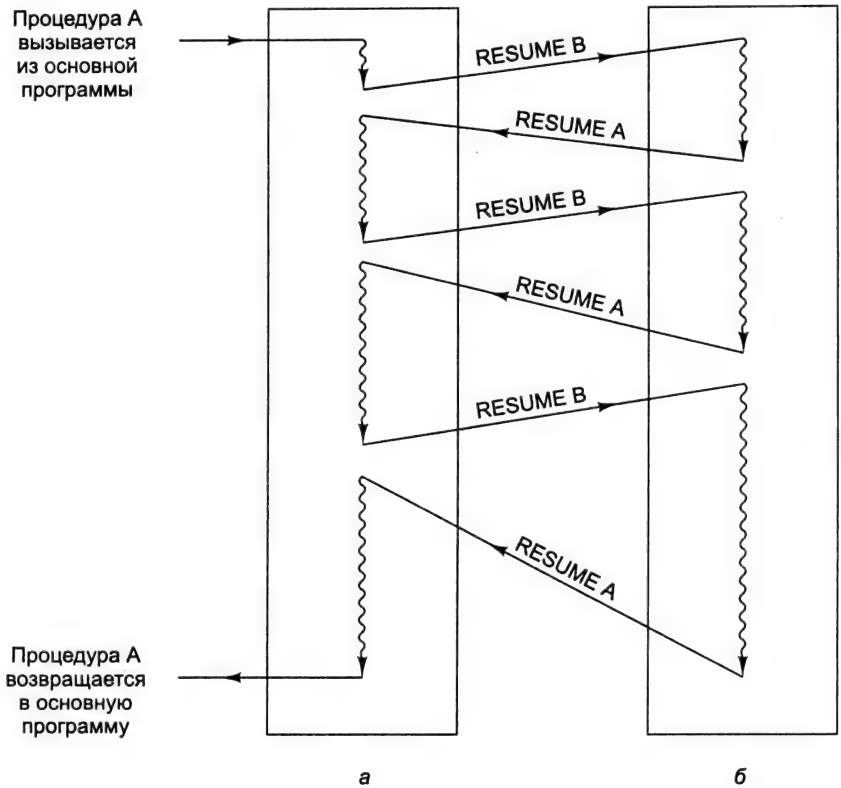


Рис. 5.29. После завершения сопрограммы выполнение начинается с того места, на котором оно завершилось в прошлый раз, а не с самого начала

Обычные команды CALL и RETURN не подходят для вызова сопрограмм, поскольку адрес для перехода берется из стека, как и при возврате, но, в отличие от возврата, при вызове сопрограммы адрес возврата помещается в определенном месте, чтобы в последующем к нему вернуться. Было бы неплохо, если бы существовала команда для замены вершины стека на счетчик команд. Эта команда сначала выталкивала бы старый адрес возврата из стека и помещала бы его во внутренний регистр, затем помещала бы счетчик команд в стек и, наконец, копировала бы содержание внутреннего регистра в счетчик команд. Поскольку одно слово выталкивается из стека, а другое помещается в стек, состояние указателя стека не меняется. Такая команда встречается очень редко, поэтому в большинстве случаев ее приходится моделировать из нескольких команд.

Ловушки

Ловушка (trap) — это особый тип вызова процедуры, который происходит при определенном условии. Обычно это очень важное, но редко встречающееся условие. Один из примеров такого условия — переполнение. В большинстве процессоров, если результат выполнения арифметической операции превышает самое боль-

шое допустимое число, срабатывает ловушка. Это значит, что поток управления переходит в какую-то фиксированную ячейку памяти, а не продолжается последовательно дальше. В этой фиксированной ячейке находится команда перехода к специальной процедуре (**обработчику системных прерываний**), которая выполняет какое-либо определенное действие, например печатает сообщение об ошибке. Если результат операции находится в пределах допустимого, ловушка не задействуется.

Важно то, что этот вид прерывания вызывается каким-то исключительным условием, вызванным самой программой и обнаруженным аппаратным обеспечением или микропрограммой. Есть и другой способ определения переполнения. Нужно иметь 1-битный регистр, который принимает значение всякий раз, когда происходит переполнение. Программист, который хочет проверить результат на переполнение, должен включить в программу явную команду «переход в случае установки бита переполнения» после каждой арифметической команды. Но это очень неудобно. А ловушки экономят время и память по сравнению с открытой проверкой под контролем программиста.

Ловушку можно реализовать путем открытой проверки, выполняемой микропрограммой (или аппаратным обеспечением). Если обнаружено переполнение, адрес ловушки загружается в счетчик команд. То, что является ловушкой на одном уровне, может находиться под контролем программы на более низком уровне. Проверка на уровне микропрограммы требует меньше времени, чем проверка под контролем программиста, поскольку она может выполняться одновременно с каким-либо другим действием. Кроме того, такая проверка экономит память, поскольку она должна присутствовать только в одном месте, например в основном цикле микропрограммы, независимо от того, сколько арифметических команд встречается в основной программе.

Наиболее распространенные условия, которые могут вызывать ловушки, — это переполнение и исчезновение значащих разрядов при операциях с плавающей точкой, переполнение при операциях с целыми числами, нарушения защиты, неопределяемый код операции, переполнение стека, попытка запустить несуществующее устройство ввода-вывода, попытка вызвать слово из ячейки с нечетным адресом и деление на 0.

Прерывания

Прерывания — это изменения в потоке управления, вызванные не самой программой, а чем-либо другим и обычно связанные с процессом ввода-вывода. Например, программа может приказать диску начать передачу информации и заставить диск произвести прерывание, как только передача данных завершится. Как и ловушка, прерывание останавливает работу программы и передает управление программе обработки прерываний, которая выполняет какое-то определенное действие. После завершения этого действия программа обработки прерываний передает управление прерванной программе. Она должна заново начать прерванный процесс в том же самом состоянии, в котором она находилась, когда произошло прерывание. Это значит, что прежнее состояние всех внутренних регистров (то есть состояние, которое было до прерывания) должно быть восстановлено.

Различие между ловушками и прерываниями в следующем: ловушки синхронны с программой, а прерывания асинхронны. Если программа перезапускается много раз с одним и тем же материалом на входе, ловушки каждый раз будут происходить в одном и том же месте, а прерывания могут меняться в зависимости от того, в какой момент человек нажимает возврат каретки. Причина воспроизводимости ловушек и невоспроизводимости прерываний состоит в том, что первые вызываются непосредственно самой программой, а прерывания вызываются программой косвенно.

Чтобы понять, как происходят прерывания, рассмотрим обычный пример: компьютеру нужно вывести на терминал строку символов. Программное обеспечение сначала собирает в буфер все символы, которые нужно вывести на экран, инициализирует глобальную переменную ptr, которая должна указывать на начало буфера, и устанавливает вторую глобальную переменную count, которая равна числу символов, выводимых на экран. Затем программное обеспечение проверяет, готов ли терминал, и если готов, то выводит на экран первый символ (например, используя регистры, которые показаны на рис. 5.30). Начав процесс ввода-вывода, центральный процессор освобождается и может запустить другую программу или сделать что-либо еще.

Через некоторое время символ отображается на экране. Теперь может начаться прерывание. Ниже перечислены основные шаги (в упрощенной форме).

Действия аппаратного обеспечения:

1. Контроллер устройства устанавливает линию прерывания на системной шине.
2. Когда центральный процессор готов к обработке прерывания, он устанавливает символ подтверждения прерывания на шине.
3. Когда контроллер устройства узнает, что сигнал прерывания был подтвержден, он помещает небольшое целое число на информационные линии, чтобы «представиться» (то есть показать, что это за устройство). Это число называется вектором прерываний[1].
4. Центральный процессор удаляет вектор прерывания с шины и временно его сохраняет.
5. Центральный процессор помещает в стек счетчик команд и слово состояния программы.
6. Затем центральный процессор определяет местонахождение нового счетчика команд, используя вектор прерывания в качестве индекса в таблице в нижней части памяти. Если, например, размер счетчика команд составляет 4 байта, тогда вектор прерываний n соответствует адресу 4n. Новый счетчик команд указывает на начало программы обслуживания прерываний для устройства, вызвавшего прерывание. Часто помимо этого загружается или изменяется слово состояния программы (например, чтобы блокировать дальнейшие прерывания).

[1] Автор не совсем прав: здесь речь должна идти о номере прерывания. Каждому типу прерывания соответствует свой номер. Термин «вектор прерываний» используется в случае, когда по номеру прерывания находится адрес программы обработки прерывания и этот адрес представляется не одним значением, а несколькими, то есть необходимо проинициализировать более одного регистра. Другими словами, адрес представляется не скалярной величиной, а многомерной, векторной. — *Примеч. научн. ред.*

Действия программного обеспечения:

1. Первое, что делает программа обработки прерываний, — сохраняет все нужные ей регистры таким образом, чтобы их можно было восстановить позднее. Их можно сохранить в стеке или в системной таблице.
2. Каждый вектор прерывания разделяется всеми устройствами данного типа, поэтому в данный момент еще не известно, какой терминал вызвал прерывание. Номер терминала можно узнать, считав значение какого-нибудь регистра.
3. Теперь можно считывать любую другую информацию о прерывании, например коды состояния.
4. Если происходит ошибка ввода-вывода, ее нужно обработать здесь.
5. Глобальные переменные ptr и count обновляются. Первая увеличивается на 1, чтобы показывать на следующий байт, а вторая уменьшается на 1, чтобы указать, что осталось вывести на 1 байт меньше. Если count все еще больше 0, значит, еще не все символы выведены на экран. Тот символ, на который в данный момент указывает ptr, копируется в выходной буферный регистр.
6. В случае необходимости выдается специальный код, который сообщает устройству или контроллеру прерывания, что прерывание обработано.
7. Восстанавливаются все сохраненные регистры.
8. Выполнение команды `RETURN FROM INTERRUPT` (выход из прерывания): возвращение центрального процессора в то состояние, в котором он находился до прерывания. После этого компьютер продолжает работу с того места, в котором ее приостановил.

С прерываниями связано важное понятие **прозрачности**. Когда происходит прерывание, производятся какие-либо действия и запускаются какие-либо программы, но когда все закончено, компьютер должен вернуться точно в то же состояние, в котором он находился до прерывания. Программа обработки прерываний, обладающая таким свойством, называется прозрачной.

Если компьютер имеет только одно устройство ввода-вывода, тогда прерывания работают точно так, как мы только что описали. Однако большой компьютер может содержать много устройств ввода-вывода, причем несколько устройств могут работать одновременно, возможно, у разных пользователей. Существует некоторая вероятность, что во время работы программы обработки прерывания другое устройство ввода-вывода тоже захочет произвести свое прерывание.

Здесь существует два подхода. Первый подход — для всех программ обработки прерываний в первую очередь (даже до сохранения регистров) предотвратить последующие прерывания. При этом прерывания будут совершаться в строгой последовательности, но это может привести к проблемам с устройствами, которые не могут долго простаивать. Например, на коммуникационной линии со скоростью передачи 9600 битов в секунду символы поступают каждые 1042 микросекунды. Если первый символ еще не обработан, когда поступает второй, то данные могут потеряться.

Если компьютер имеет подобные устройства ввода-вывода, то лучше всего приписать каждому устройству определенный приоритет, высокий для более критич-

ных и низкий для менее критичных устройств. Центральный процессор тоже должен иметь приоритеты, которые определяются по одному из полей слова состояния программы. Если устройство с приоритетом n вызывает прерывание, программа обработки прерывания тоже должна работать с приоритетом n.

Если работает программа обработки прерываний с приоритетом n, любая попытка другого устройства с более низким приоритетом будет игнорироваться, пока программа обработки прерываний не завершится и пока центральный процессор не возвратится к выполнению программы более низкого приоритета. С другой стороны, прерывания, поступающие от устройств с более высоким приоритетом, должны происходить без задержек.

Поскольку сами программы обработки прерываний подвержены прерыванию, лучший способ строгого управления — сделать так, чтобы все прерывания были прозрачными. Рассмотрим простой пример с несколькими прерываниями. Компьютер имеет три устройства ввода-вывода: принтер, диск и линию RS232 с приоритетами 2, 4 и 5 соответственно. Изначально (t=0; t — время) работает пользовательская программа. Вдруг при t=10 принтер совершает прерывание. Запускается программа обработки прерывания принтера, как показано на рис. 5.30.



Рис. 5.30. Пример с несколькими прерываниями. Последовательность действий

При t=15 линия RS232 порождает сигнал прерывания. Так как линия RS232 имеет более высокий приоритет (5), чем принтер (2), прерывание происходит. Состояние машины, при котором работает программа обработки прерывания принтера, сохраняется в стеке, и начинается выполнение программы обработки прерывания RS232.

Немного позже, при t=20, диск завершает свою работу. Однако его приоритет (4) ниже, чем приоритет работающей в данный момент программы обработки прерыва-

ний (5), поэтому центральный процессор не подтверждает прием сигнала прерывания, и диск вынужден простаивать. При t=25 заканчивается программа RS232, и машина возвращается в то состояние, в котором она находилась до прерывания RS232, то есть в то состояние, когда работала программа обработки прерывания принтера с приоритетом 2. Как только центральный процессор переключается на приоритет 2, еще до того как будет выполнена первая команда, диск с приоритетом 4 совершает прерывание и запускается программа обработки прерываний диска. После ее завершения снова продолжается программа обработки прерываний принтера. Наконец, при t=40 все программы обработки прерываний завершаются и выполнение пользовательской программы начинается с того места, на котором она прервалась.

Со времен процессора 8088 все процессоры Intel имеют два уровня (приоритета) прерываний: маскируемые и немаскируемые прерывания. Немаскируемые прерывания обычно используются только для сообщения об очень серьезных ситуациях, например об ошибках четности в памяти. Все устройства ввода-вывода используют одно маскируемое прерывание.

Когда устройство ввода-вывода вызывает прерывание, центральный процессор использует вектор прерывания при индексировании таблицы из 256 элементов, чтобы найти адрес программы обработки прерываний. Элементы таблицы представляют собой 8-байтные дескрипторы сегмента. Таблица может начинаться в любом месте памяти. Глобальный регистр указывает на ее начало.

При наличии только одного уровня прерываний центральный процессор не может сделать так, чтобы устройство с более высоким приоритетом прерывало работу программы обработки прерываний с более низким приоритетом и чтобы устройство с более низким приоритетом не смогло прерывать выполнение программы обработки прерываний с более высоким приоритетом. Для решения этой проблемы центральные процессоры Intel обычно используют внешний контроллер прерываний (например, 8259A). При первом прерывании (например, с приоритетом n) работа процессора приостанавливается. Если после этого происходит еще одно прерывание с более высоким приоритетом, контроллер прерывания вызывает прерывание во второй раз. Если второе прерывание обладает более низким приоритетом, оно не реализуется до окончания первого. Чтобы эта система работала, контроллер прерывания должен каким-либо образом узнавать о завершении текущей программы обработки прерываний. Поэтому когда текущее прерывание полностью обработано, центральный процессор должен посылать специальную команду контроллеру прерываний.

Ханойская башня

Теперь, когда мы уже изучили уровень команд трех машин, нам нужно все обобщить. Давайте подробно рассмотрим тот же пример программы («Ханойская башня») для всех трех машин. В листинге 5.6 приведена версия этой программы на языке Java. В следующих разделах для решения этой задачи мы предложим программы на ассемблере.

Однако вместо того, чтобы давать трансляцию версии на языке Java, для машин Pentium II и UltraSPARC II мы представим трансляцию версии на языке C,

чтобы избежать проблем с вводом-выводом Java. Единственное различие — это замена оператора Java printf на стандартный оператор языка C

```
printf("Переместить диск с %d на %d\n", i,j)
```

Синтаксис строки printf не важен (строка печатается буквально за исключением `%d` — это означает, что следующее целое число будет дано в десятичной системе счисления). Здесь важно только то, что процедура вызывается с тремя параметрами: форматирующей строкой и двумя целыми числами.

Мы использовали язык C для Pentium II и UltraSPARC II, поскольку библиотека ввода-вывода Java не доступна для этих машин, а библиотека C доступна. Для JVM мы будем использовать язык Java. Разница минимальна: всего один оператор вывода строки на экран.

Решение задачи «Ханойская башня» на ассемблере Pentium II

В листинге 5.7 приведен возможный вариант трансляции программы на языке C для компьютера Pentium II. Регистр EBP используется в качестве указателя фрейма. Первые два слова применяются для установления связи, поэтому первый параметр n (или N, поскольку регистр для макроассемблера не важен) находится в ячейке EBP+8, а за ним следуют параметры i и j в ячейках EBP+12 и EBP+16 соответственно. Локальная переменная k находится в EBP+20.

Листинг 5.7. Решение задачи «Ханойская башня» для машины Pentium II

```
            .586                ;компилируется для Pentium
.MODEL FLAT
PUBLIC _towers                  ;экспорт 'towers'
EXTERN _printf:NEAR             ;импорт printf
.CODE
_towers:   PUSH EBP             ;сохраняет EBP (указатель фрейма)
           MOV EBP, ESP         ;устанавливает новый указатель фрейма над ESP
           CMP[EBP+8],1         ;if(n==1)
           JNE L1               ;переход, если n?1
           MOV EAX, [EBP+16] ;printf("…", i, j);
           PUSH EAX             ;сохранение параметров i, j и формата
           MOV EAX, [EBP+12] ;строка помещается в стек
           PUSH EAX             ;в обратном порядке. Таково требование языка C
           PUSH OFFSET FLAT:format ; OFFSET FLAT – это адрес формата
           CALL _printf         ;вызов процедуры printf
           ADD ESP, 12          ;удаление параметров из стека
           JMP Done             ;завершение
L1:        MOV EAX,6            ;начало вычисления k=6-i-j
           SUB EAX, [EBP+12] ;EAX=6-i
           SUB EAX, [EBP+16] ;EAX=6-i-j
           MOV [EBP+20], EAX ;k=EAX
           PUSH EAX             ;начало процедуры towers(n-1, i, k)
           MOV EAX, [EBP+12] ;EAX=i
           PUSH EAX             ;помещает в стек i
           MOV EAX, [EBP+8]  ;EAX=n
           DEC EAX;EAX=n-1
           PUSH EAX             ;помещает в стек n-1
           CALL _towers         ;вызов процедуры towers(n-1, i, 6-i-j)
```

```
        ADD ESP, 12       ;удаление параметров из стека
        MOV EAX, [EBP+16] ;начало процедуры towers (1, i, j)
        PUSH EAX          ;помещает в стек j
        MOV EAX, [EBP+12] ;EAX=i
        PUSH EAX          ;помещает в стек i
        PUSH 1            ;помещает в стек 1
        CALL _towers      ;вызывает процедуру towers(1, i, j)
        ADD ESP, 12       ;удаляет параметры из стека
        MOV EAX, [EBP+12] ;начало процедуры towers(n-1, 6-i-j, i)
        PUSH EAX          ;помещает в стек i
        MOV EAX, [EBP+20] ;EAX=k
        PUSH EAX          ;помещает в стек k
        MOV EAX, [EBP+8]  ;EAX = n
        DEC EAX;EAX=n-1
        PUSH EAX          ;помещает в стек n-1
        CALL _towers      ;вызов процедуры towers(n-1, 6-i-j, i)
        ADD ESP, 12       ;корректировка указателя  стека
Done:   LEAVE             ;подготовка к выходу
        RET 0             ;возврат к вызывающей программе
.DATA
format  DB "Переместить диск с %d на %d\n" ;форматирующая строка
END
```

Процедура начинается с создания нового фрейма в конце старого. Для этого значение регистра ESP копируется в указатель фрейма EBP. Затем n сравнивается с 1, и если n>1, то совершается переход к оператору `else`. Тогда программа `then` помещает в стек три значения: адрес форматирующей строки, i и j, и вызывает саму себя.

Параметры помещаются в стек в обратном порядке, поскольку это требуется для программ на языке C. Необходимо поместить указатель на форматирующую строку в вершину стека. Процедура `printf` имеет переменное число параметров, и если параметры будут помещаться в стек в прямом порядке, то процедура не сможет узнать, в каком месте стека находится форматирующая строка.

После вызова процедуры к регистру ESP прибавляется 12, чтобы удалить параметры из стека. На самом деле они не удаляются из памяти, но корректировка (изменение) регистра ESP делает их недоступными через обычные операции со стеком.

Выполнение части `else` начинается с L1. Здесь сначала вычисляется выражение 6–i–j, и полученное значение сохраняется в переменной k. Сохранение значения в переменной k избавляет от необходимости вычислять это во второй раз.

Затем процедура вызывает сама себя три раза, каждый раз с новыми параметрами. После каждого вызова стек освобождается.

Рекурсивные процедуры иногда приводят людей в замешательство. Но на самом деле они совсем несложные. Просто параметры помещаются в стек, и вызывается процедура.

Решение задачи «Ханойская башня» на ассемблере UltraSPARC II

А теперь рассмотрим то же самое для UltraSPARC II. Программа приведена в листинге 5.8. Поскольку программа для UltraSPARC II совершенно нечитаема даже после длительных тренировок, мы решили определить несколько символов, чтобы

прояснить дело. Чтобы такая программа работала, ее перед ассемблированием нужно пропустить через программу под названием cpp (препроцессор C). Здесь мы используем строчные буквы, поскольку ассемблер Pentium II требует этого (это на тот случай, если читатели захотят напечатать и запустить эту программу).

Листинг 5.8. Решение задачи «Ханойская башня» для UltraSPARC II

```
#define N %i0          /* N – это входной параметр 0 */
#define I %i1          /* I – это входной параметр 1 */
#define J %i2          /* J – это входной параметр 2 */
#define K %l0          /* K – это локальная переменная 0 */
#define Param0 %o0     /* Param0 – это выходной параметр 0 */
#define Param1 %o1     /* Param1 – это выходной параметр 1 */
#define Param2 %o2     /* Param2 – это выходной параметр 2 */
#define Scratch %l1    /*примеч.: cpp использует запись комментариев как в языке C*/
          .proc 04
          .global towers
towers:   save %sp,-112, %sp
          cmp N, 1                          ! if(n== 1)
          bne Else                          ! if (n != 1) goto Else
          sethi %hi(format), Param0         ! printf("Переместить диск с %d на %d\n", i, j)
          or Param0, %lo(format), Param0    ! Param0 = адрес форматирующей строки
          mov I, Param1                     ! Param1 = i
          call printf                       ! вызов printf ДО установки параметра 2 (j)
          mov J, Param2                     ! пустая операция для установки параметра 2
          b Done                            ! завершение
          nop                               ! вставляет пустую операцию

Else:     mov 6, K                          ! начало вычисления k = 6 -i-j
          sub K,J,K                         ! k=6-j
          sub K,I,K                         !k=6-i-j

          add N, -1, Scratch    ! начало процедуры towers(n-1, i, k)
          mov Scratch, Param0   ! Scratch = n-1
          mov I, Param1         ! параметр 1 = i
          call towers           ! вызов процедуры towers ДО установки параметра 2 (k)
          mov K, Param2         ! пустая операция после вызова процедуры для установки
                                  параметра2

          mov 1, Param0         ! начало процедуры towers(1, i, j)
          mov I, Param1         ! параметр1=i
          call towers           ! вызов процедуры towers ДО установки параметра 2 (j)
          mov J, Param2         ! параметр 2 = j

          mov Scratch, Param0   ! начало процедуры towers(n-1, k, j)
          mov K, Param1         ! параметр 1 = k
          call towers           ! вызов процедуры towers ДО установки параметра 2 (j)
          mov J, Param2         ! параметр 2 = j
Done:     ret                   ! выход из процедуры
          restore               ! вставка пустой команды после  ret для восстановления окон
format:   .asciz "Переместить диск с %d на %d\n"
```

По алгоритму версия UltraSPARC идентична версии Pentium II. В обоих случаях сначала проверяется n, и если n>1, то совершается переход к `else`. Основные

сложности версии UltraSPARC II связаны с некоторыми свойствами архитектуры команд.

Сначала UlraSPARC II должен передать адрес форматирующей строки в `printf`, но машина не может просто переместить адрес в регистр, который содержит выходящий параметр, поскольку нельзя поместить 32-битную константу в регистр за одну команду. Для этого требуется выполнить две команды: `SETHI` и `OR`.

После вызова не нужно делать подстройку стека, поскольку регистровое окно корректируется командой `RESTORE` в конце процедуры. Возможность помещать выходящие параметры в регистры и не обращаться к памяти дает огромный выигрыш в производительности.

А теперь рассмотрим команду `NOP`, которая следует за Done. Это пустая операция. Эта команда всегда будет выполняться, даже если она следует за командой условного перехода. Сложность состоит в том, что процессор UltraSPARC II сильно конвейеризирован, и к тому моменту, когда аппаратное обеспечение обнаруживает команду перехода, следующая команда уже практически закончена. Добро пожаловать в прекрасный мир программирования `RISC`!

Эта особенность распространяется и на вызовы процедур. Рассмотрим первый вызов процедуры `towers` в части `else`. Процедура помещает n–1 в %o0, а i — в %o1, но совершает вызов процедуры `towers` до того, как поместит последний параметр в нужное место. На компьютере Pentium II вы сначала передаете параметры, а затем вызываете процедуру. А здесь вы сначала передаете часть параметров, затем вызываете процедуру, и только после этого передаете последний параметр. К тому моменту, когда машина осознает, что она имеет дело с командой `CALL`, следующую команду все равно приходится выполнять (из-за конвейеризации системы). А почему бы в этом случае не использовать пустую операцию, чтобы передать последний параметр? Даже если самая первая команда вызванной процедуры использует этот параметр, он уже будет на своем месте.

Наконец, рассмотрим часть команды `Done`. Здесь после команды `RET` тоже вставляется пустая операция. Эта пустая операция используется для команды `RESTORE`, которая увеличивает на 1 значение CWP, чтобы вернуть регистровое окно в прежнее состояние.

Решение задачи «Ханойская башня» на ассемблере для JVM

Соответствующая программа дана в листинге 5.9. Решение довольно простое, за исключением процесса ввода-вывода. Эта программа была порождена компилятором Java, переделана в символический язык ассемблера и обработана определенным образом для удобочитаемости. Компилятор JVM хранит три параметра n, i и j в локальных переменных 0, 1 и 2 соответственно. Локальная переменная k хранится в локальной переменной 3. Ко всем четырем локальным переменным можно обратиться с помощью 1-байтного кода операции, например `ILOAD_0`. В результате двоичная версия этой программы в JVM получается очень короткой (всего 67 байтов).

Листинг 5.9. Решение задачи «Ханойская башня» для JVM

```
ILOAD_0           // лок. переменная 0 = n; помещает в стек n
ICONST_1          // помещает в стек 1
IFJCMPNE L1       //if(n!=1)gotoL1
GETSTATIC #13     // n == 1; эта команда обрабатывает выражение println
NEW #7            // размещает буфер для строки, которую нужно создать
DUP               // дублирует указатель на буфер
LDC #2            // помещает в стек указатель на цепочку "перенести диск с"
INVOKESPECIAL #10 // копирует эту цепочку в буфер
ILOAD_1           // помещает в стек i
INVOKEVIRTUAL #11 // превращает i в цепочку и присоединяет к новому буферу
LDC #1            // помещает в стек указатель на цепочку "на"
INVOKEVIRTUAL #12 // присоединяет эту цепочку к буферу
ILOAD_2           // помещает в стек j
INVOKEVIRTUAL #11 // превращает j в цепочку и присоединяет ее к буферу
INVOKEVIRTUAL #15 // преобразование строки
INVOKEVIRTUAL #14 // вызов println
RETURN            // выход из процедуры towers
L1: BIPUSH6       // Часть Else: вычисление k = 6-i-j
ILOAD_1           // лок. переменная 1 = i; помещает в стек i
ISUB              // вершина стека = 6-i
ILOAD_2           // лок. переменная 2= j; помещает в стек j
ISUB              // вершина стека = 6-i-j
ISTORE_3          // лок. перем. 3 = k = 6-i-j; стек сейчас пуст
ILOAD_0           // начало работы процедуры towers(n-1,i, k);помещает в стек n
ICONST_1          // помещает в стек 1
ISUB              // вершина стека = n-1
ILOAD_1           // помещает в стек i
ILOAD_3           // помещает в стек k
INVOKESTATIC #16  // вызывает процедуру towers(n-1, i, k)
ICONST_1          // начинается работа процедуры towers(1, i, j);помещает в стек 1
ILOAD_1           // помещает в стек i
ILOAD_2           // помещает в стек j
INVOKESTATIC #16  // вызов процедуры towers(1, i, j)
ILOAD_0 "         // начало работы процедуры towers(n-1, k, j); помещает в стек n
ICONST_1          // помещает в стек 1
ISUB              // вершина стека = n-1
ILOAD_3           // помещает в стек k
ILOAD_2           // помещает в стек j
INVOKESTATIC #16  // вызов процедуры towers(n-1, k, j)
RETURN            // выход из процедуры towers
```

Сначала программа помещает в стек параметр n и константу 1, а затем сравнивает их с помощью команды IF_ICMPNE. Эта команда обратна команде IF_ICMPEQ, которая используется в машине IJVM. Она выталкивает из стека два операнда и совершает переход, если они различны.

Если они одинаковы, выполнение программы продолжается последовательно. Следующие 13 команд определяют буфер строки и строят в нем цепочку, которая затем передается в println для вывода на экран. После завершения печати совершается выход из процедуры.

Если говорить кратко, эти 13 команд размещают буфер строки в «кучу» и заполняют его. Команда GETSTATIC индексирует набор констант, чтобы получить слово 13, которое содержит указатель на дескриптор для буфера строки. Команда NEW использует этот дескриптор для размещения буфера строки в «куче». Следую-

щие 11 команд связывают две цепочки и два целых числа в одну цепочку в этом буфере и передают ее в `println` для вывода на экран.

Если n не равно 1, управление передается к L1. Переменная k вычисляется простым путем с использованием арифметических операций над числами в стеке. Затем совершаются три вызова один за другим.

Машина IJVM содержит ряд команд для вызова процедур. В данном случае компилятор использовал три разные команды. Все они содержат 2-байтный операнд, который индексирует набор констант, чтобы найти указатель на дескриптор, сообщающий все о вызываемой процедуре. Константы #10, #11 и т. д. — индексы в наборе констант.

Наличие нескольких типов команд для вызова процедур связано с оптимизацией. Команда `INVOKESTATIC` используется для вызова статических процедур, например `towers`. Команда `INVOKESPECIAL` применяется для вызова процедур инициализации, нестандартных процедур и процедур надкласса текущего класса. Наконец, команда `INVOKEVIRTUAL` используется для внутренних (библиотечных) вызовов.

Intel IA-64

Со временем увеличивать скорость работы IA-32 становилось все сложнее и сложнее. Единственным возможным решением этой проблемы стала разработка совершенно новой архитектуры команд. Новая архитектура, которая разрабатывалась совместно компаниями Intel и Hewlett Packard, получила название **IA-64**. Это полностью 64-битная машина от начала до конца. Появление полной серии процессоров, в которых реализуется эта архитектура, ожидается в ближайшие годы. Самым первым процессором этого типа был процессор **Merced** с высокой производительностью, хотя в будущем наверняка появится полный спектр процессоров разного уровня.

Поскольку все, что делает компания Intel, очень важно для компьютерной промышленности, мы подробно рассмотрим архитектуру IA-64. Однако ключевые идеи этой архитектуры уже очень хорошо известны многим исследователям, поэтому они могут отражаться в других разработках. Вообще, некоторые из них уже реализованы в разных формах в экспериментальных системах. В следующих разделах мы изложим, с какой проблемой столкнулась компания Intel, каким образом архитектура IA-64 помогла справиться с этой проблемой и как работают ключевые идеи этого проекта.

Проблема с Pentium II

Основная проблема заключалась в том, что IA-32 — это старая архитектура команд с совершенно не подходящими для современной техники свойствами. Это архитектура `CISC` с командами разной длины и огромным количеством различных форматов, которые трудно декодировать быстро и на лету. Современная техника лучше всего работает с архитектурами команд `RISC` с командами одной

длины и с кодом операции фиксированной длины, который легко декодировать. Команды IA-32 можно разбить на микрооперации типа RISC во время выполнения программы, но для этого требуется дополнительное аппаратное обеспечение (пространство на микросхеме), что занимает время и усложняет разработку. Это первый недостаток.

IA-32 — это архитектура, которая ориентирована на двухадресные команды. В настоящее время популярны архитектуры команд типа загрузка/сохранение, где обращение к памяти совершается только в тех случаях, когда нужно поместить операнды в регистры, а все вычисления выполняются с использованием трехадресных регистровых команд. Поскольку скорость работы процессора растет гораздо быстрее, чем скорость работы памяти, положение дел с IA-32 со временем все больше ухудшается. Это второй недостаток.

Архитектура IA-32 содержит небольшой и нерегулярный набор регистров. Из-за столь малого числа регистров общего назначения (четыре или шесть, в зависимости от того, как считать ESI и EDI) постоянно нужно записывать в память промежуточные результаты, и поэтому приходится делать дополнительные обращения к памяти, даже когда они по логике вещей не нужны. Это третий недостаток.

Из-за недостаточного числа регистров возникает множество ситуаций зависимостей, особенно WAR-зависимостей, поскольку промежуточные результаты нужно куда-то поместить, а дополнительных регистров нет. При недостатке регистров требуется переименование регистров в скрытые регистры. Во избежание слишком частых промахов кэш-памяти команды приходится выполнять не по порядку. Однако семантика архитектуры IA-32 определяет точные прерывания, поэтому команды, выполняемые не по порядку, должны записывать результаты в выходные регистры в строгом порядке. Для всего этого требуется очень сложное аппаратное обеспечение. Это четвертый недостаток.

Чтобы скорость работы была высокой, нужна сильно конвейеризированная система (12 стадий). Однако это значит, что для выполнения команды потребуется 11 циклов. Следовательно, становится существенным точное предсказание переходов, поскольку в конвейер должны попадать только нужные команды. Но даже при низком проценте неправильных предсказаний существенно снижается производительность. Это пятый недостаток.

Чтобы избежать проблем с неправильным прогнозированием переходов, процессору приходится осуществлять спекулятивное выполнение команд со всеми вытекающими отсюда последствиями. Это шестой недостаток.

Мы не будем перечислять недостатки дальше, поскольку уже сейчас ясно, что за ними кроется реальная проблема. И мы еще не упомянули, что 32-битные адреса архитектуры IA-32 ограничивают размер отдельных программ до 4 Гбайт, а это требование очень сложно выполнять на дорогостоящих серверах с высокой производительностью.

Ситуации с IA-32 можно сравнить с положением в небесной механике как раз перед появлением Коперника. В те времена в астрономии доминировала теория, что Земля является центром Вселенной и неподвижна, а планеты движутся вокруг нее. Однако новые наблюдения показывали все больше и больше несоответствий этой теории действительности, и в конце концов теория полностью разрушилась.

Компания Intel находится приблизительно в таком же положении. Огромное количество транзисторов в процессоре Pentium II предназначено для переделывания команд CISC в команды RISC, разрешения конфликтов, прогнозирования переходов, исправления неправильных предсказаний и решения многих других задач подобного рода, оставляя лишь незначительное число транзисторов на долю реальной работы, которая нужна пользователю. Поэтому компания Intel пришла к следующему выводу: нужно выбросить IA-32 и начать все заново (IA-64).

Модель IA-64: открытое параллельное выполнение команд

Первой реализацией архитектуры IA-32 был 64-битный процессор RISC (один из примеров этого процессора — UltraSPARC II). Поскольку архитектура IA-64 была разработана совместно с компанией Hewlett Packard, в ее основу, несомненно, легла архитектура PA-RISC. Merced — это двухрежимный процессор, который может выполнять и программы IA-32, и программы IA-64, но мы будем говорить только об архитектуре IA-64.

Архитектура IA-64 — это архитектура типа загрузка/сохранение с 64-битными адресами и 64-битными регистрами. Здесь имеется 64 регистра общего назначения, доступных для программ IA-64 (и дополнительные регистры для программ IA-32). Все команды имеют фиксированный формат: код операции, два 6-битных поля для указания входных регистров, одно 6-битное поле для указания выходного регистра и еще одно 6-битное поле, которое мы обсудим позже. Большинство команд берут два регистровых операнда, выполняют над ними какую-нибудь операцию и помещают результат в выходной регистр. Для параллельного выполнения различных операций существует много функциональных блоков. Как видим, пока ничего необычного в этой архитектуре нет. Большинство RISC-процессоров имеют сходную архитектуру.

А необычной здесь является идея о **пучке** связанных команд. Команды поступают группами по три штуки (рис. 5.31). Каждая такая группа называется пучком. Каждый 128-битный пучок содержит три 40-битные команды фиксированного формата и 8-битный шаблон. Пучки могут быть связаны вместе (при этом используется бит конца пучка), поэтому в одном пучке может присутствовать более трех команд. Формат содержит информацию о том, какие команды могут выполняться параллельно. При такой системе и при наличии большого числа регистров компилятор может выделять блоки команд и сообщать процессору, что эти команды можно выполнять параллельно. Таким образом, компилятор должен переупорядочивать команды, проверять, нет ли взаимозависимостей, проверять доступность функциональных блоков и т. д. вместо аппаратного обеспечения. Основная идея состоит в том, что работа упорядочивания и распределения команд RISC передается от аппаратного обеспечения компилятору. Именно поэтому эта технология называется **EPIC (Explicitly Parallel Instruction Computing — технология параллельной обработки команд с явным параллелизмом)**.

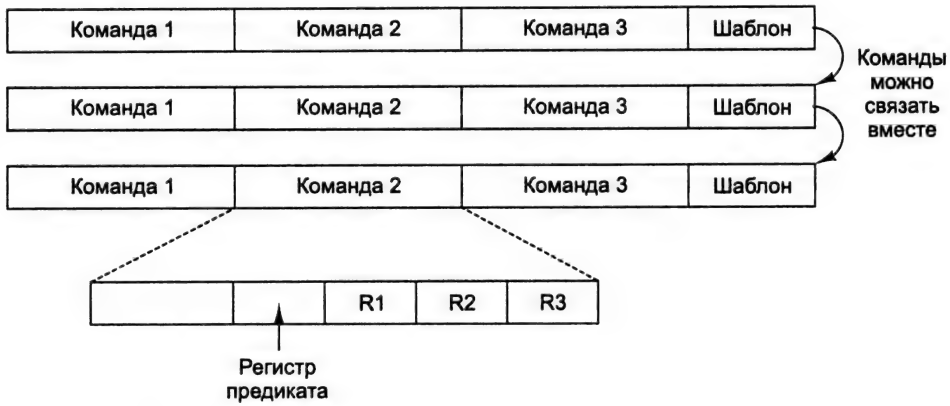


Рис. 5.31. Архитектура IA-64 основана на пучках из трех команд

Есть несколько причин, по которым следует совершать упорядочивание команд во время компиляции. Во-первых, поскольку теперь всю работу выполняет компилятор, аппаратное обеспечение можно сильно упростить, используя миллионы транзисторов для других полезных функций (например, можно увеличить кэш-память первого уровня). Во-вторых, для любой программы распределение должно производиться только один раз (во время компиляции). В-третьих, поскольку компилятор теперь выполняет всю работу, у поставщика программного обеспечения появится возможность использовать компилятор, который часами оптимизирует свою программу.

Идея пучков команд может быть использована при создании целого семейства процессоров. Процессоры с низкой производительностью могут запускать по одному пучку за цикл. Перед тем как выпустить новый пучок, центральный процессор должен дождаться завершения всех команд. Процессоры с высокой производительностью способны запускать несколько пучков за один цикл (по аналогии с современными суперскалярными процессорами). Кроме того, процессор с высокой производительностью может начать запускать команды из нового пучка еще до того, как закончится выполнение всех команд предыдущего пучка. Естественно, нужно все время проверять, доступны ли нужные регистры и функциональные блоки, но зато вовсе не надо проверять, не будут ли конфликтовать разные команды одного пучка, поскольку компилятор гарантирует, что этого не произойдет.

Предикация

Еще одна особенность архитектуры IA-64 — новый способ обработки условных переходов. Если бы была возможность избавиться от большинства из них, центральный процессор стал бы гораздо проще и работал бы гораздо быстрее. На первый взгляд может показаться, что устранить условные переходы невозможно, поскольку в программах всегда полно операторов if. Однако в архитектуре IA-64 используется специальная технология, названная **предикацией**[1], которая позволяет сильно сократить их число [10, 63]. Ниже мы кратко опишем эту технологию.

[1] От англ. predicate — утверждение, предикат. — *Примеч. научн. ред.*

В современных машинах все команды являются безусловными в том смысле, что когда центральный процессор натыкается на команду, он просто ее выполняет. Здесь никогда не решается вопрос: «Выполнять или не выполнять?» Напротив, в архитектуре с предикацией команды содержат условия, которые сообщают, в каком случае нужно выполнять команду, а в каком — нет. Именно этот сдвиг от безусловных команд к командам с предикацией позволяет избавиться от многих условных переходов. Вместо того чтобы выбирать ту или иную последовательность безусловных команд, все команды сливаются в одну последовательность команд с предикацией, используя разные предикаты для различных команд.

Чтобы понять, как работает предикация, рассмотрим простой пример (листинги 5.10–5.12), в котором показано **условное выполнение** команд (условное выполнение — предтеча предикации). В листинге 5.10 мы видим оператор if. В листинге 5.11 мы видим трансляцию этого оператора в три команды: команду сравнения, команду условного перехода и команду перемещения.

Листинг 5.10. Оператор if

```
if (R1==0)
      R2=R3;
```

Листинг 5.11. Код на ассемблере для листинга 5.10

```
      CMP R1,0
      BNE L1
      MOV R2,R3
L1:
```

Листинг 5.12. Условная команда

```
CMOVZ R2,R3,R1
```

В листинге 5.12 мы избавились от условного перехода, используя новую команду `CMOVZ`, которая является условным перемещением. Эта команда проверяет, равен ли третий регистр R1 нулю. Если да, то команда копирует R3 в R2. Если нет, то команда не выполняет никаких действий.

Если у нас есть команда, которая может копировать данные, когда какой-либо регистр равен нулю, значит, у нас может быть и такая команда, которая копирует данные, если какой-нибудь регистр не равен нулю. Пусть это будет команда `CMOVN`. При наличии обеих команд мы уже на пути к полному условному выполнению. Представим оператор `if` с несколькими присваиваниями в части `then` и несколькими присваиваниями в части `else`. Весь этот кусок программы можно транслировать в код, который будет устанавливать какой-нибудь регистр на 0, если условие не выполнено, и на какое-нибудь другое значение, если условие выполнено. Следуя установке регистров, присваивания в части `then` можно скомпилировать в последовательность команд `CMOVN`, а присваивания в части `else` — в последовательность команд `CMOVZ`.

Все эти команды, регистровая установка, `CNOVN` и `CMOVZ` формируют единый основной блок без условных переходов. Команды можно даже переупорядочить при компиляции или во время выполнения программы. Единственное требование при этом состоит в том, чтобы условие было известно к тому моменту, когда условные команды уже нужно помещать в выходные регистры (то есть где-то

в конце конвейера). Простой пример куска программы с then и else приведен в листингах 5.13–5.15.

Листинг 5.13. Оператор if

```
if(R1==0) {
R2=R3;
R4=R5;
} else {
R6=R7;
R8=R9;
}
```

Листинг 5.14. Код на ассемблере для листинга 5.13

```
       CMP R1,0
       BNE L1
       MOV R2,R3
       MOV R4,R5
       BR L2
L1: MOV R6,R7
       MOV R8,R9
L2:
```

Листинг 5.15. Условное выполнение

```
CMOVZ R2,R3,R1
CMOVZ R4,R5,R1
CMOVN R6,R7,R1
CMOVN R8,R9,R1
```

Мы показали только очень простые условные команды (взятые из Pentium II), но в архитектуре IA-64 все команды предикатны. Это значит, что выполнение каждой команды можно сделать условным. Дополнительное 6-битное поле, о котором мы упомянули выше, выбирает один из 64 1-битных предикатных регистров. Следовательно, оператор if может быть скомпилирован в код, который устанавливает один из предикатных регистров на 1, если условие истинно, и на 0, если условие ложно. Одновременно с этим данное поле автоматически устанавливает другой предикатный регистр на обратное значение. При использовании предикации машинные команды, которые формируют операторы then и else, будут сливаться в единый поток команд, первый из них — с использованием предиката, а второй — с использованием его обратного значения.

Листинги 5.16–5.18 показывают, как можно использовать предикацию для устранения переходов. Команда CMPEQ сравнивает два регистра и устанавливает предикатный регистр P4 на 1, если они равны, и на 0, если они не равны. Кроме того, команда устанавливает парный регистр, например P5, на обратное условие. Теперь команды частей if и then можно поместить одну за другой, причем каждая из них связывается с каким-нибудь предикатным регистром (регистр указывается в угловых скобках). Сюда можно поместить любой код, при условии что каждая команда предсказывается правильно.

Листинг 5.16. Оператор if

```
if(R1==R2)
       R3=R4+R5;
Else
       R6=R4-R5
```

Листинг 5.17. Код на ассемблере для листинга 5.16

```
        CMP R1,R2
        BNE L1
        MOV R3,R4
        ADD R3,R5
        BR L2
L1: MOV R6,R4
        SUB R6,R5
L2:
```

Листинг 5.18. Выполнение с предикацией

```
CMPEQ R1,R2,P4
<P4> ADD R3,R4,R5
<P5> SUB R6,R4,R5
```

В архитектуре IA-64 эта идея доведена до логического конца: здесь с предикатными регистрами связаны и команды сравнения, и арифметические команды, а также некоторые другие команды, выполнение которых зависит от какого-либо предикатного регистра. Команды с предикацией могут помещаться в конвейер последовательно без каких-либо проблем и простаиваний. Поэтому они очень полезны.

В архитектуре IA-64 предикация происходит следующим образом. Каждая команда действительно выполняется, и в самом конце конвейера, когда уже нужно сохранять результат в выходной регистр, производится проверка, истинно ли предсказание. Если да, то результаты просто записываются в выходной регистр. Если предсказание ложно, то записи в выходной регистр не происходит. Подробно о предикации вы можете прочитать в книге [34].

Спекулятивная загрузка

Еще одна особенность IA-64 — наличие спекулятивной загрузки. Если команда LOAD спекулятивна и оказывается ложной, то вместо того чтобы вызвать исключение (exception), она просто останавливается и сообщает, что регистр, с которым она связана, недействителен. Если этот регистр будет использоваться позднее, то произойдет исключение (exception).

Компилятор должен перемещать команды LOAD в более ранние позиции относительно других команд с тем, чтобы они выполнялись еще до того, как они понадобятся. Поскольку выполнение этих команд начинается раньше, чем нужно, они могут завершиться еще до того, как потребуются результаты. Компилятор вставляет команду CHECK в том месте, где ему нужно получить значение определенного регистра. Если значение там уже есть, команда CHECK работает как NOP, и выполнение программы сразу продолжается дальше. Если значения в регистре еще нет, следующая команда должна простаивать.

Итак, машина с архитектурой IA-64 имеет несколько источников повышения скорости. Во-первых, это современная конвейеризированная трехадресная машина RISC типа загрузка/сохранение. Во-вторых, компилятор определяет, какие команды могут выполняться одновременно, не вступая в конфликт, и группирует эти команды в пучки. Таким образом, процессор может просто распределять пучок, не совершая никаких проверок. В-третьих, предикация позволяет сливать команды обоих переходов от оператора if, устраняя при этом условный переход, а также и само прогнозирование этого перехода. Наконец, спекулятивная загрузка позволя-

ет вызывать операнды заранее, и даже если позднее окажется, что эти операнды не нужны, ничего страшного не произойдет.

Проверка в реальных условиях

Если все эти нововведения функционируют, процессор Merced действительно будет чрезвычайно мощным. Однако здесь нужно высказать несколько предостережений. Во-первых, такая продвинутая машина никогда не создавалась раньше. История показывает, что грандиозные планы часто не удается осуществить по самым разным причинам.

Во-вторых, придется составлять компиляторы для IA-64. А составление хорошего компилятора для IA-64 — дело очень сложное. Кроме того, многочисленные исследования в параллельном программировании, проведенные за последние 30 лет, оказались не очень успешными. Если в программе есть какой-то параллелизм или если компилятор не может извлечь его, все пучки команд в архитектуре IA-64 будут короткими, а от этого большого увеличения производительности не произойдет.

В-третьих, для реализации этой идеи должна существовать полностью 64-битная операционная система. Windows 95 и Windows 98 такими системами не являются и вряд ли когда-нибудь будут. Это значит, что всем придется перейти на Windows NT или UNIX, и такой переход не будет безболезненным. Спустя 10 лет с момента появления 32-битного процессора 386 система Windows 95 все еще содержала множество 16-битных команд и никогда не использовала аппаратное обеспечение с сегментацией, которое около 10 лет включается во все процессоры Intel (сегментацию мы будем обсуждать в главе 6). Сколько времени потребуется на то, чтобы операционные системы стали полностью 64-битными, никто не знает.

В-четвертых, многие будут судить об архитектуре IA-64 по тому, как на ней будут работать старые 16-битные игры MS DOS. Когда появился Pentium Pro, он не пользовался особенным успехом, поскольку старые 16-битные программы работали на нем с такой же скоростью, как и на обычной машине Pentium. Поскольку старые 16-битные (и 32-битные) программы не будут использовать новые особенности процессоров IA-64, никакие предикатные регистры им не помогут. Людям придется приобретать средства наращивания для своего программного обеспечения, подходящие для новых компиляторов типа IA-64. Многие из них будут очень дорогими.

Наконец, другие производители (включая Intel) могут выпустить альтернативные варианты, которые будут давать высокую производительность, используя при этом более привычные архитектуры RISC, возможно, с большим количеством условных команд. Более высокоскоростные версии Pentium II также будут серьезным соперником для IA-64. Вероятно, пройдет очень много лет, прежде чем IA-64 будет доминировать на компьютерном рынке, подобно архитектуре IA-32.

Краткое содержание главы

Для большинства людей уровень архитектуры команд — это «машинный язык». На этом уровне машина имеет память с байтовой организацией или с пословной

организацией, состоящую из нескольких десятков мегабайтов и содержащую такие команды, как MOVE, ADD и BEQ.

В большинстве современных компьютеров память организована в виде последовательности байтов, при этом 4 или 8 байтов группируются в слова. Обычно в машине имеется от 8 до 32 регистров, каждый из которых содержит одно слово.

Команды обычно имеют 1, 2 или 3 операнда, обращение к которым происходит с помощью различных способов адресации: непосредственной, прямой, регистровой, индексной и т. д. Команды обычно могут перемещать данные, выполнять унарные и бинарные операции (в том числе арифметические и логические), совершать переходы, вызывать процедуры, осуществлять циклы, а иногда и выполнять некоторые операции ввода-вывода. Типичные команды перемещают слово из памяти в регистр или наоборот, складывают, вычитают, умножают или делят два регистра или регистр и слово из памяти, или сравнивают два значения в регистрах или памяти. Обычно в компьютере содержится не более 200 команд.

Для осуществления передачи управления на втором уровне используется ряд элементарных действий: переходы, вызовы процедур, вызовы сопрограмм, ловушки и прерывания. Переходы нужны для того, чтобы остановить одну последовательность команд и начать новую. Процедуры нужны для того, чтобы изолировать какой-то блок программы, который можно вызывать из различных мест этой же программы. Сопрограммы позволяют двум потокам управления работать одновременно. Ловушки используются для сообщения об исключительных ситуациях (например, о переполнении). Прерывания позволяют осуществлять процесс ввода-вывода параллельно с основным вычислением, при этом центральный процессор получает сигнал, как только ввод-вывод завершен.

Задачу «Ханойская башня» можно решить с использованием рекурсии.

Наконец, архитектура IA-64 использует модель вычисления EPIC. Для повышения скорости работы в этой архитектуре предусмотрены предикация и спекулятивная загрузка. IA-64 может иметь значительное преимущество над Pentium II, но она возлагает на компилятор огромное бремя параллелизма.

Вопросы и задания

1. В Pentium II команды могут содержать любое число байтов, даже нечетное. В UltraSPARC II все команды содержат четное число байтов. В чем преимущество системы Pentium II?
2. Разработайте расширенный код операций, который позволяет закодировать в 36-битной команде следующее:
   - 7 команд с двумя 32-битными адресами и номером одного 3-битного регистра;
   - 500 команд с одним 15-битным адресом и номером одного 3-битного регистра;
   - 50 команд без адресов и регистров.

3. Можно ли разработать такой расширенный код операций, который позволял бы кодировать в 12-битной команде следующее:
   - 4 команды с тремя регистрами;
   - 255 команд с одним регистром;
   - 16 команд без регистров.

   (Размер регистра составляет 3 бита.)
4. В некоторой машине имеются 16-битные команды и 6-битные адреса. Одни команды содержат один адрес, другие — два. Если существует n двухадресных команд, то каково максимальное число одноадресных команд?
5. Имеется одноадресная машина с регистром-аккумулятором. Ниже приведены значения некоторых слов в памяти:
   - слово 20 содержит число 40;
   - слово 30 содержит число 50;
   - слово 40 содержит число 60;
   - слово 50 содержит число 70.

   Какие значения следующие команды загрузят в регистр-аккумулятор?
   - `LOAD IMMEDIATE 20`
   - `LOAD DIRECT 20`
   - `LOAD INDIRECT 20`
   - `LOAD IMMEDIATE 30`
   - `LOAD DIRECT 30`
   - `LOAD INDIRECT 30`.
6. Для каждого из четырех видов машин — безадресной, одноадресной, двухадресной и трехадресной — напишите программу вычисления следующего выражения:

   X=(A+B×C)/(D−E×F).
7. В наличии имеются следующие команды:
   - безадресные:

     ```
     PUSH M
     POP M
     ADD
     SUB
     MUL
     DIV
     ```
   - одноадресные:

     ```
     LOAD M
     STORE M
     ADD M
     ```

```
SUB M
MUL M
DIV M
```

- двухадресные:

```
MOV (X=Y)
ADD (X=X+Y)
SUB (X=X-Y)
MUL (X=X*Y)
DIV (X=X/Y)
```

- трехадресные:

```
MOV (X=Y)
ADD (X=Y+Z)
SUB (X=Y-Z)
MUL (X=Y*Z)
DIV (X=Y/Z).
```

8. M — это 16-битный адрес памяти, а X, Y и Z — это или 16-битные адреса, или 4-битные регистры. Безадресная машина использует стек, одноадресная машина использует регистр-аккумулятор, а оставшиеся две имеют 16 регистров и команды, которые оперируют со всеми комбинациями ячеек памяти и регистров. Команда `SUB X,Y` вычитает Y из X, а команда `SUB X,Y,Z` вычитает Z из Y и помещает результат в X. Если длина кодов операций равна 8 битам, а размеры команд кратны 4 битам, сколько битов нужно каждой машине для вычисления X?
9. Придумайте такой механизм адресации, который позволяет определять в 6-битном поле произвольный набор из 64 адресов, не обязательно смежных.
10. В чем недостаток самоизменяющихся программ, которые не были упомянуты в тексте?
11. Переделайте следующие формулы из инфиксной записи в обратную польскую запись:
    - A+B+C+D+E
    - (A+B)×(C+D)+E
    - (A×B)+(C×D)+E
    - (A−B)×(((C−D×E)/F)/G)×H
12. Переделайте следующие формулы из обратной польской записи в инфиксную запись:
    - A B + C + D ×
    - A B / C D / +
    - A B C D E + × × /
    - A B C D E × F /+ G−H/×+

13. Какие из следующих пар формул в обратной польской записи математически эквивалентны?
    - A B + C + и A B C ++
    - A B – C – и A B C – –
    - A B × C+ и A B C +×
14. Напишите три формулы в обратной польской записи, которые нельзя переделать в инфиксную запись.
15. Переделайте следующие инфиксные логические формулы в обратной польской записи.
    - (A И B) ИЛИ C
    - (A ИЛИ B) И (A ИЛИ C)
    - (A И B) ИЛИ (C И D)
16. Переделайте следующую инфиксную формулу в обратную польскую запись и напишите код JVM, чтобы выполнить ее.

    (2×3+4)–(4/2+1)
17. Команда языка ассемблера

    ```
    MOV REG, ADDR
    ```

    означает загрузку регистра из памяти компьютера Pentium II. Однако для UltraSPARC II для загрузки регистра из памяти нужно написать

    ```
    LOAD ADDR, REG
    ```

    Почему порядок операндов разный?
18. Сколько регистров содержится в машине, форматы команд которой даны на рис. 5.16?
19. В форматах команд на рис. 5.16 бит 23 используется для различения формата 1 и формата 2. Однако для определения формата 3 никакого специального бита не предусмотрено. Как аппаратное обеспечение узнает, что нужно использовать формат 3?
20. Обычно программа определяет местонахождение переменной X в пределах интервала от A до B. Если бы имелась трехадресная команда с операндами A, B и X, сколько битов кода условия было бы установлено этой командой?
21. Pentium II содержит бит кода условия, который следит за переносом бита 3 после выполнения арифметической операции. Зачем это нужно?
22. В UltraSPARC II нет такой команды, которая загружает в регистр 32-битное число. Вместо нее обычно используется последовательность из двух команд: `SETHI` и `ADD`. Существуют ли еще какие-нибудь способы загрузки 32-битного числа в регистр? Аргументируйте.
23. Один из ваших друзей стучится к вам в комнату в 3 часа ночи и радостно сообщает, что у него появилась замечательная идея: команда с двумя кодами операций. Что вы сделаете в этой ситуации: отправите своего друга получать патент или пошлете его обратно к чертежной доске?

24. В программировании очень распространены следующие формы проверки:

```
if (n==0)…
if (i>j)…
if (k<4)…
```

Предложите команду, которая будет проверять эти условия эффективно. Какие поля имеются в вашей команде?

25. Покажите для 16-битного двоичного числа 1001 0101 1100 0011:
    - Сдвиг вправо на 4 бита с заполнением нулями.
    - Сдвиг вправо на 4 бита с расширением по знаку.
    - Сдвиг влево на 4 бита.
    - Циклический сдвиг влево на 4 бита.
    - Циклический сдвиг вправо на 4 бита.
26. Как можно в машине, в которой нет команды `CLR`, очистить слово памяти?
27. Вычислите логическое выражение (A И B) ИЛИ C для:
    - A=1101 0000 1010 1101
    - B=1111 1111 0000 1111
    - C=0000 0000 0010 0000
28. Придумайте, как поменять местами две переменные A и B, не используя при этом третью переменную или регистр. Подсказка: подумайте о команде ИСКЛЮЧАЮЩЕЕ ИЛИ.
29. На некотором компьютере можно перемещать число из одного регистра в другой, сдвигать каждый из них влево на разное количество байтов и складывать полученные результаты за меньшее время, чем потребовалось бы для выполнения умножения. При каком условии эта последовательность команд будет полезна для вычисления произведения «константа × переменная»?
30. Разные машины имеют разную плотность команд (то есть разное число байтов, которое требуется для выполнения определенного вычисления). Транслируйте следующие три фрагмента программы на языке Java на ассемблер для Pentium II, UltraSPARC II и JVM. Затем посчитайте, сколько байтов требуется для выполнения каждого выражения для каждой машины (предполагается, что i и j — это локальные переменные памяти):
    - i=3;
    - i=j;
    - i=j–1;
31. В этой главе рассматривались команды цикла для работы с циклами `for`. Разработайте команду для обращения с циклами `while`.
32. Предположим, что ханойские монахи могут перемещать один диск за 1 минуту (они не торопятся закончить работу, поскольку в Ханое очень мало вакансий для людей с подобными навыками). Сколько времени им потребуется, чтобы решить задачу (то есть переместить все 64 диска)? Ответ дайте в годах.

33. Почему устройства ввода-вывода помещают вектор прерывания на шину? Разве нельзя вместо этого сохранить соответствующую информацию в таблице в памяти?
34. Компьютер для считывания информации с диска использует канал прямого доступа к памяти. Диск содержит 64 сектора по 512 байтов на дорожке. Время оборота диска 16 мс. Ширина шины 16 битов. Каждая передача шины занимает 500 нс. В среднем для одной команды процессора требуется два цикла шины. Насколько скорость работы процессора замедляется из-за прямого доступа к памяти?
35. Почему программам обработки прерываний приписываются определенные приоритеты, а обычные процедуры приоритетов не имеют?
36. Архитектура IA-64 содержит необычайно большое число регистров (64). Связано ли столь большое количество регистров с использованием предикации? Если да, то каким образом? Если нет, то зачем тогда их так много?
37. В пятой главе обсуждалось понятие спекулятивной загрузки. Но о командах спекулятивного сохранения мы не упоминали. Почему? Может быть, они просто аналогичны спекулятивным загрузкам, или существует какая-то другая причина, по которой мы не стали о них говорить?
38. Когда нужно связать две локальные сети, между ними помещается мост, связанный с обеими сетями. Каждый передаваемый какой-либо сетью пакет вызывает прерывание на мосту, чтобы мост мог определить, нужно ли этот пакет пересылать. Предположим, что на обработку прерывания и проверку пакета требуется 250 мкс, но пересылка этого пакета в случае необходимости совершается с использованием прямого доступа в память, поэтому центральный процессор не загружается. Если все пакеты вмещают 1 Кбайт, то какова максимальная скорость передачи данных на каждой из сетей?
39. На рис. 5.24 указатель фрейма указывает на первую локальную переменную. Какая информация нужна программе, чтобы выйти из процедуры и вернуться в исходное положение?
40. Напишите подпрограмму на языке ассемблера для превращения целого двоичного числа со знаком в код ASCII.
41. Напишите подпрограмму на языке ассемблера для превращения инфиксной формулы в обратную польскую запись.
42. «Ханойская башня» — это не единственная рекурсивная процедура, любимая многими компьютерщиками. Есть еще одна очень популярная рекурсивная процедура n!, где n!=n(n−1)! Подчиняется ограничивающему условию 0!=1. Напишите на вашем любимом языке ассемблера процедуру для вычисления n!.
43. Попробуйте решить задачу «Ханойская башня» без использования рекурсии путем содержания стека в массиве. Предупреждаем, что, вероятно, вы не сможете найти решения.

Глава 6
Уровень операционной системы

Как мы уже говорили, современный компьютер состоит из ряда уровней, каждый из которых добавляет дополнительные функции к уровню, который находится под ним. Мы рассмотрели цифровой логический уровень, микроархитектурный уровень и уровень команд. Настало время перейти к следующему уровню — уровню операционной системы.

Операционная система — это программа, которая добавляет ряд команд и особенностей к тем, которые обеспечиваются уровнем команд. Обычно операционная система реализуется главным образом в программном обеспечении, но нет никаких веских причин, по которым ее нельзя было бы реализовать в аппаратном обеспечении (как микропрограммы). Уровень операционной системы показан на рис. 6.1.

Хотя и уровень операционной системы, и уровень команд абстрактны (в том смысле, что они не являются реальным аппаратным обеспечением), между ними есть важное различие. Набор команд уровня операционной системы — это полный набор команд, доступных для прикладных программистов. Он содержит практически все команды более низкого уровня, а также новые команды, которые добавляет операционная система. Эти новые команды называются **системными вызовами**. Они вызывают определенную службу операционной системы, в частности одну из ее команд. Обычный системный вызов считывает какие-нибудь данные из файла.

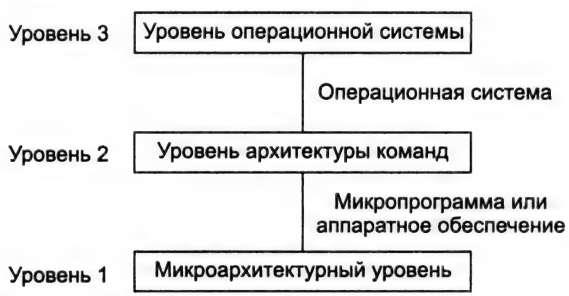


Рис. 6.1. Расположение уровня операционной системы

Уровень операционной системы всегда интерпретируется. Когда пользовательская программа выполняет команду операционной системы, например чтение данных из файла, операционная система выполняет эту команду шаг за шагом, точно

так же, как микропрограмма выполняет команду ADD. Однако когда программа выполняет команду уровня архитектуры команд, эта команда выполняется непосредственно микроархитектурным уровнем без участия операционной системы.

В этой книге мы можем лишь очень кратко в общих чертах рассказать вам об уровне операционной системы. Мы сосредоточимся на трех важных особенностях. Первая особенность — это виртуальная память. Виртуальная память используется многими операционными системами. Она позволяет создать впечатление, что у машины больше памяти, чем есть на самом деле. Вторая особенность — файл ввода-вывода. Это понятие более высокого уровня, чем команды ввода-вывода, которые мы рассматривали в предыдущей главе. Третья особенность — параллельная обработка (как несколько процессов могут выполняться, обмениваться информацией и синхронизироваться). Понятие процесса является очень важным, и мы подробно рассмотрим его ниже в этой главе. Под процессом можно понимать работающую программу и всю информацию об ее состоянии (о памяти, регистрах, счетчике команд, вводе-выводе и т. д.). После обсуждения этих основных характеристик мы покажем, как они применяются к операционным системам двух машин из трех наших примеров: Pentium II (Windows NT) и UltraSPARC II (UNIX). Поскольку picoJava II обычно используется для встроенных систем, у этой машины нет операционной системы.

Виртуальная память

В первых компьютерах память была очнь мала по объему и к тому же дорого стоила. IBM-650, ведущий компьютер того времени (конец 50-х годов), содержал всего 2000 слов памяти. Один из первых 60 компиляторов, ALGOL, был написан для компьютера с размером памяти всего 1024 слова. Древняя система с разделением времени прекрасно работала на компьютере PDP-1, общий размер памяти которого составлял всего 4096 18-битных слов для операционной системы и пользовательских программ. В те времена программисты тратили очень много времени на то, чтобы впихнуть свои программы в крошечную память. Часто приходилось использовать алгоритм, который работает намного медленнее другого алгоритма, поскольку лучший алгоритм был слишком большим по размеру и программа, в которой использовался этот лучший алгоритм, могла не поместиться в память компьютера.

Традиционным решением этой проблемы было использование вспомогательной памяти (например, диска). Программист делил программу на несколько частей, так называемых **оверлеев**, каждый из которых помещался в память. Чтобы прогнать программу, сначала следовало считывать и запускать первый оверлей. Когда он завершался, нужно было считывать и запускать второй оверлей и т. д. Программист отвечал за разбиение программы на оверлеи и решал, в каком месте вспомогательной памяти должен был храниться каждый оверлей, контролировал передачу оверлеев между основной и вспомогательной памятью и вообще управлял всем этим процессом без какой-либо помощи со стороны компьютера.

Хотя эта технология широко использовалась на протяжении многих лет, она требовала длительной кропотливой работы, связанной с управлением оверлеями. В 1961 году группа исследователей из Манчестера (Англия) предложила метод автоматического выполнения процесса наложения, при котором программист мог вообще не знать об этом процессе [42]. Этот метод, который сейчас называется **виртуальной памятью**, имел очевидное преимущество, поскольку освобождал программиста от огромного количества нудной работы. Впервые этот метод был использован в ряде компьютеров, выпущенных в 60-е годы. К началу 70-х годов виртуальная память появилась в большинстве компьютеров. В настоящее время даже компьютеры[1] на одной микросхеме, в том числе Pentium II и UltraSPARC II, содержат очень сложные системы виртуальной памяти. Мы рассмотрим их ниже в этой главе.

Страничная организация памяти

Группа ученых из Манчестера выдвинула идею о разделении понятий адресного пространства и адресов памяти. Рассмотрим в качестве примера типичный компьютер того времени с 16-битным полем адреса в командах и 4096 словами памяти. Программа, работающая на таком компьютере, могла обращаться к 65536 словам памяти (поскольку адреса были 16-битными, а 2^{16}=65536). Обратите внимание, что число адресуемых слов зависит только от числа битов адреса и никак не связано с числом реально доступных слов в памяти. **Адресное пространство** такого компьютера состоит из чисел 0, 1, 2, ..., 65535, так как это набор всех возможных адресов. Однако в действительности компьютер мог иметь гораздо меньше слов в памяти.

До изобретения виртуальной памяти приходилось проводить жесткое различие между теми адресами, которые меньше 4096, и теми, которые равны или больше 4096. Эти две части рассматривались как полезное адресное пространство и бесполезное адресное пространство соответственно (адреса выше 4095 были бесполезными, поскольку они не соответствовали реальным адресам памяти). Никакого различия между адресным пространством и адресами памяти не проводилось, поскольку между ними подразумевалось взаимно-однозначное соответствие.

Идея разделения понятий адресного пространства и адресов памяти состоит в следующем. В любой момент времени можно получить прямой доступ к 4096 словам памяти, но это не значит, что они непременно должны соответствовать адресам памяти от 0 до 4095. Например, мы могли бы сообщить компьютеру, что при обращении к адресу 4096 должно использоваться слово из памяти с адресом 0, при обращении к адресу 4097 — слово из памяти с адресом 1, при обращении к адресу 8191 — слово из памяти с адресом 4095 и т. д. Другими словами, мы определили отображение из адресного пространства в действительные адреса памяти (рис. 6.2).

[1] Строго говоря, сверхбольшие интегральные микросхемы, на которых сейчас располагают микропроцессоры, в том числе и упоминаемый автором процессор Pentium II, не являются компьютерами. Компьютер должен содержать помимо процессора память и контроллер управления ею, устройства ввода-вывода и соответствующие контроллеры для управления ими. — *Примеч. научн. ред.*

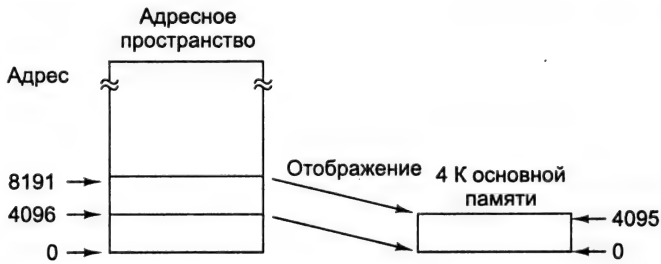


Рис. 6.2. Виртуальные адреса памяти с 4096 по 8191 отображаются в адреса основной памяти с 0 по 4095

Возникает интересный вопрос: а что произойдет, если программа совершит переход в один из адресов с 8192 по 12287? В машине без виртуальной памяти произойдет ошибка, на экране появится фраза «Несуществующий адрес памяти» и выполнение программы остановится. В машине с виртуальной памятью будет иметь место следующая последовательность шагов:

1. Слова с 4096 до 8191 будут размещены на диске.
2. Слова с 8192 до 12287 будут загружены в основную память.
3. Отображение адресов изменится: теперь адреса с 8192 до 12287 соответствуют ячейкам памяти с 0 по 4095.
4. Выполнение программы будет продолжаться, как будто ничего ужасного не случилось.

Такая технология автоматического наложения называется **страничной организацией памяти**, а куски программы, которые считываются с диска, называются **страницами**.

Возможен и другой, более сложный способ отображения адресов из адресного пространства в реальные адреса памяти. Адреса, к которым программа может обращаться, мы будем называть **виртуальным адресным пространством**, а реальные адреса памяти в аппаратном обеспечении — **физическим адресным пространством**. Схема распределения памяти или таблица страниц соотносит виртуальные адреса с физическими адресами. Предполагается, что на диске достаточно места для хранения полного виртуального адресного пространства (или, по крайней мере, той его части, которая используется в данный момент).

Программы пишутся так, как будто в основной памяти хватит места для размещения всего виртуального адресного пространства, даже если это не соответствует действительности. Программы могут загружать слова из виртуального адресного пространства или записывать слова в виртуальное адресное пространство, несмотря на то, что на самом деле физической памяти для этого не хватает. Программист может писать программы, даже не осознавая, что виртуальная память существует. Просто создается такое впечатление, что объем памяти данного компьютера достаточно велик.

Позднее мы сопоставим страничную организацию памяти с процессом сегментации, при котором программист должен знать о существовании сегментов. Еще

раз подчеркнем, что страничная организация памяти создает иллюзию большой линейной основной памяти такого же размера, как адресное пространство. В действительности основная память может быть меньше (или больше), чем виртуальное адресное пространство. То, что память большого размера просто моделируется с помощью страничной организации памяти, нельзя определить по программе (только с помощью контроля синхронизации). При обращении к любому адресу всегда появляются требующиеся данные или нужная команда. Поскольку программист может писать программы и при этом ничего не знать о существовании страничной организации памяти, этот механизм называют **прозрачным**.

Ситуация, когда программист использует какой-либо виртуальный механизм и даже не знает, как он работает, не нова. В архитектуры команд, например, часто включается команда `MUL` (команда умножения), даже если в аппаратном обеспечении нет специального устройства для умножения. Иллюзия, что машина может перемножать числа, поддерживается микропрограммой. Точно так же операционная система может создавать иллюзию, что все виртуальные адреса поддерживаются реальной памятью, даже если это неправда. Только разработчикам операционных систем и тем, кто изучает операционные системы, нужно знать, как создается такая иллюзия.

Реализация страничной организации памяти

Для виртуальной памяти требуется диск для хранения полной программы и всех данных. Естественно, при изменении копии должен меняться и оригинал.

Виртуальное адресное пространство разбивается на ряд страниц равного размера, обычно от 512 до 64 Кбайт, хотя иногда встречается 4 Мбайт. Размер страницы всегда должен быть степенью двойки. Физическое адресное пространство тоже разбивается на части равного размера таким образом, чтобы каждая такая часть основной памяти вмещала ровно одну страницу. Эти части основной памяти называются **страничными кадрами**. На рисунке 6.2 основная память содержит только один страничный кадр. На практике обычно имеется несколько тысяч страничных кадров.

На рисунке 6.3, *а* показан один из возможных вариантов разделения первых 64 K виртуального адресного пространства на страницы по 4 K (отметим, что сейчас мы говорим о 64 K и 4 K адресов). Адрес может быть байтом, но может быть словом в компьютере, в котором последовательно расположенные слова имеют последовательные адреса. Виртуальную память, изображенную на рис. 6.3, можно реализовать посредством таблицы страниц, в которой количество элементов равно количеству страниц в виртуальном адресном пространстве. Здесь для простоты мы показали только первые 16 элементов. Когда программа пытается обратиться к слову из первых 64 K виртуальной памяти, чтобы вызвать команду или данные или чтобы сохранить данные, сначала она порождает виртуальный адрес от 0 до 65532 (предполагается, что адреса слов должны делиться на 4). Для порождения этого адреса могут использоваться любые стандартные способы адресации, в том числе индексирование и косвенная адресация.



Страница Виртуальный адрес

15	61440 P 65535
14	57344 P 61439
13	53248 P 57343
12	49152 P 53247
11	45056 P 49151
10	40960 P 45055
9	36864 P 40959
8	32768 P 36863
7	28672 P 32767
6	24576 P 28671
5	20480 P 24575
4	16384 P 20479
3	12288 P 16383
2	8192 P 12287
1	4096 P 8191
0	0 P 4095

а

Нижние 32 К адресов основной памяти

Страничный кадр	Физические адреса
7	28672 P 32767
6	24576 P 28671
5	20480 P 24575
4	16384 P 20479
3	12288 P 16383
2	8192 P 12287
1	4096 P 8191
0	0 P 4095

б

Рис. 6.3. Первые 64 К виртуального адресного пространства разделены на 16 страниц по 4 К каждая (*а*); 32 К основной памяти разделены на 8 страничных кадров по 4 К каждый (*б*)

На рис. 6.3, *б* изображена физическая память, состоящая из восьми страничных кадров по 4 К. Эту память можно ограничить до 32 К, поскольку: 1) это вся память машины (для процессора, встроенного в стиральную машину или микроволновую печь, этого достаточно), или 2) оставшаяся часть памяти занята другими программами.

А теперь рассмотрим, как можно 32-битный виртуальный адрес отобразить на физический адрес основной памяти. В конце концов, память воспринимает только реальные адреса и не воспринимает виртуальные, поэтому такое отображение должно быть сделано. Каждый компьютер с виртуальной памятью содержит устройство для осуществления отображения виртуальных адресов на физические. Это устройство называется **контроллером управления памятью (MMU – Memory Management Unit)**. Он может находиться на микросхеме процессора или на отдельной микросхеме рядом с процессором. В нашем примере контроллер управления памятью отображает 32-битный виртуальный адрес в 15-битный физический адрес, поэтому ему требуется 32-битный входной регистр и 15-битный выходной регистр.

Чтобы понять, как работает контроллер управления памятью, рассмотрим пример на рис. 6.4. Когда в контроллер управления памятью поступает 32-битный виртуальный адрес, он разделяет этот адрес на 20-битный номер виртуальной страницы и 12-битное смещение внутри этой страницы (поскольку страницы в нашем примере по 4 К). Номер виртуальной страницы используется в качестве индекса в таблице страниц для нахождения нужной страницы. На рис. 6.4 номер виртуальной страницы равен 3, поэтому из таблицы выбирается элемент 3.

Сначала контроллер управления памятью проверяет, находится ли нужная страница в текущий момент в памяти. Поскольку у нас есть 2^{20} виртуальных страниц и

всего 8 страничных кадров, не все виртуальные страницы могут находиться в памяти одновременно. Контроллер управления памятью проверяет **бит присутствия** в данном элементе таблицы страниц. В нашем примере этот бит равен 1. Это значит, что страница в данный момент находится в памяти.

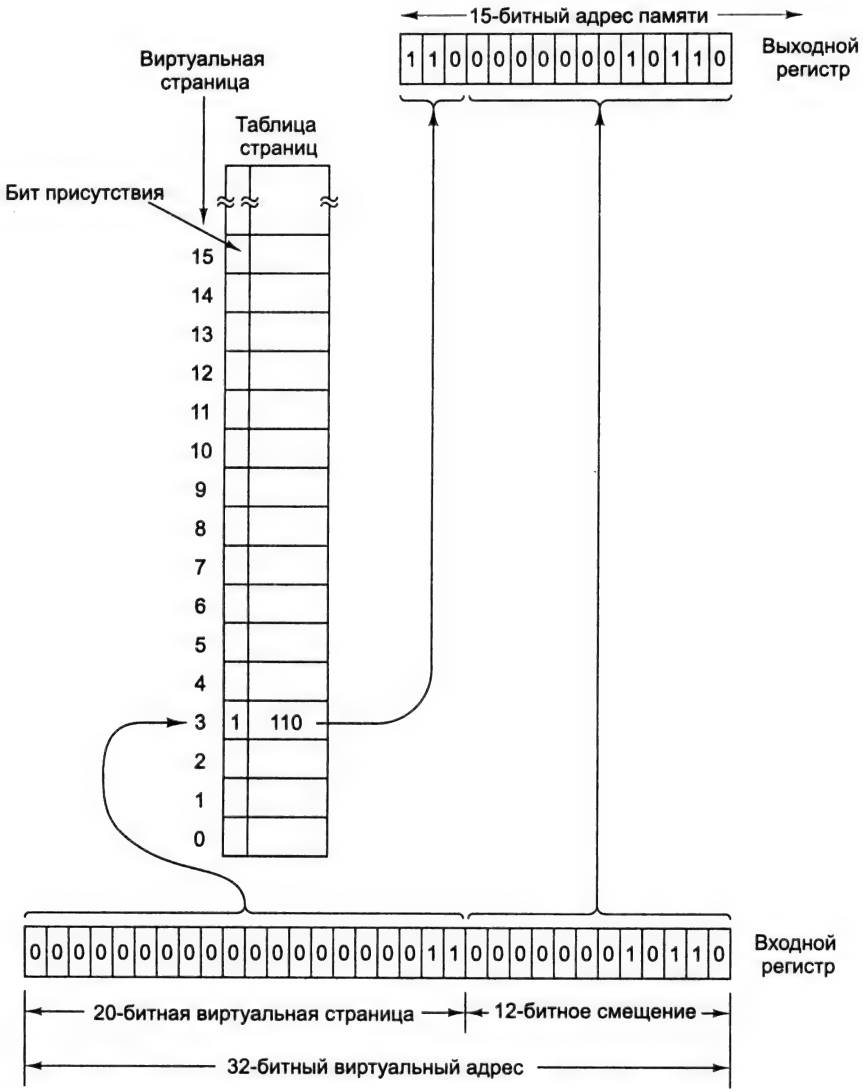


Рис. 6.4. Формирование адреса основной памяти из адреса виртуальной памяти

Далее из выбранного элемента таблицы нужно взять значение страничного кадра (в нашем примере — 6) и скопировать его в старшие три бита 15-битного выходного регистра. Нужно именно три бита, потому что в физической памяти находится 8 страничных кадров. Параллельно с этой операцией младшие 12 битов виртуального адреса (поле смещения страницы) копируются в младшие 12 битов выходного

регистра. Затем полученный 15-битный адрес отправляется в кэш-память или основную память для поиска.

На рисунке 6.5 показано возможное отображение виртуальных страниц в физические страничные кадры. Виртуальная страница 0 находится в страничном кадре 1. Виртуальная страница 1 находится в страничном кадре 0. Виртуальной страницы 2 нет в основной памяти. Виртуальная страница 3 находится в страничном кадре 2. Виртуальной страницы 4 нет в основной памяти. Виртуальная страница 5 находится в страничном кадре 6 и т. д.

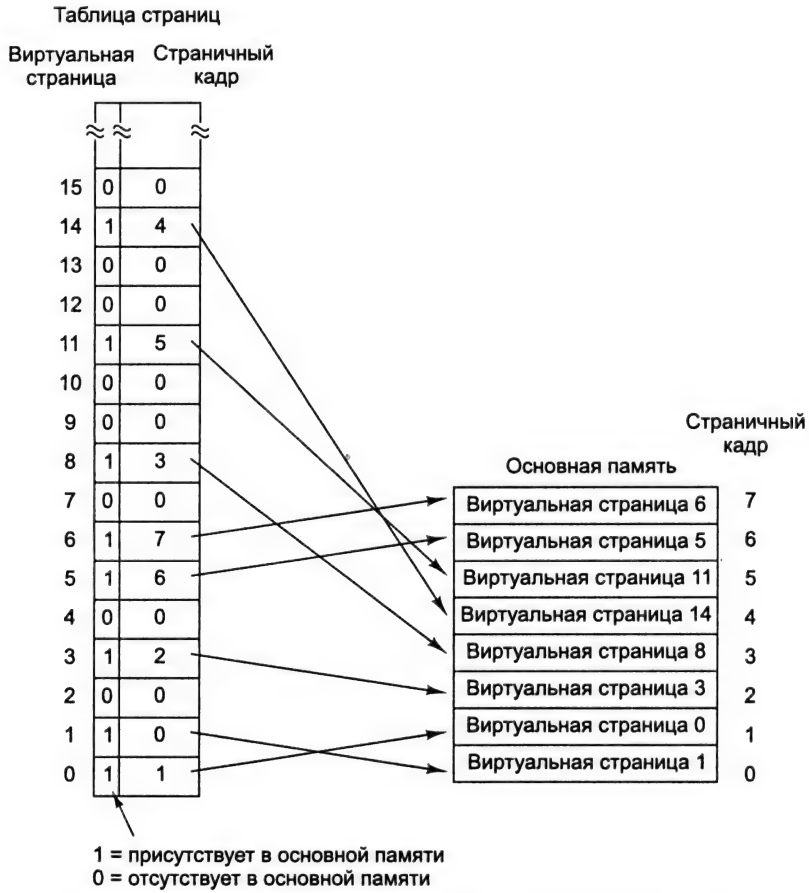


Рис. 6.5. Возможное отображение первых 16 виртуальных страниц в основную память, содержащую 8 страничных кадров

Вызов страниц по требованию и рабочее множество

Ранее предполагалось, что виртуальная страница, к которой происходит обращение, находится в основной памяти. Однако это предположение не всегда верно, поскольку в основной памяти недостаточно места для всех виртуальных страниц.

При обращении к адресу страницы, которой нет в основной памяти, происходит **ошибка из-за отсутствия страницы**. В случае такой ошибки операционная система должна считать нужную страницу с диска, ввести новый адрес физической памяти в таблицу страниц, а затем повторить команду, которая вызвала ошибку.

На машине с виртуальной памятью можно запустить программу даже в том случае, если ни одной части программы нет в основной памяти. Просто таблица страниц должна указывать, что абсолютно все виртуальные страницы находятся во вспомогательной памяти. Если центральный процессор пытается вызвать первую команду, он сразу получает ошибку из-за отсутствия страницы, в результате чего страница, содержащая первую команду, загружается в память и вносится в таблицу страниц. После этого начинается выполнение первой команды. Если первая команда содержит два адреса и оба эти адреса находятся на разных страницах, причем обе страницы не являются страницей, в которой находится команда, то произойдет еще две ошибки из-за отсутствия страницы и еще две страницы будут перенесены в основную память до завершения команды. Следующая команда может вызвать еще несколько ошибок и т. д.

Такой метод работы с виртуальной памятью называется **вызовом страниц по требованию**. Он похож на один из способов кормления младенцев: когда младенец кричит, вы его кормите (в противоположность кормлению по расписанию). При вызове страниц по требованию страницы переносятся в основную память только в случае необходимости, но не заранее.

Вопрос о том, стоит ли использовать вызов страниц по требованию или нет, имеет смысл только в самом начале при запуске программы. Когда программа проработает некоторое время, нужные страницы уже будут собраны в основной памяти. Если это компьютер с разделением времени и процессы откачиваются обратно, проработав примерно 100 миллисекунд, то каждая программа будет запускаться много раз. Для каждой программы распределение памяти уникально и при переключении с одной программы на другую меняется, поэтому в системах с разделением времени такой подход не годится.

Альтернативный подход основан на наблюдении, что большинство команд обращаются к адресному пространству не равномерно. Обычно большинство обращений относятся к небольшому числу страниц. При обращении к памяти можно вызвать команду, вызвать данные или сохранить данные. В каждый момент времени t существует набор страниц, которые использовались при последних k обращениях. Деннинг [29] назвал этот набор страниц **рабочим множеством**.

Поскольку рабочее множество обычно меняется очень медленно, можно, опираясь на последнее перед остановкой программы рабочее множество, предсказать, какие страницы понадобятся при новом запуске программы. Эти страницы можно загрузить заранее перед очередным запуском программы (предполагается, что предсказание будет точным).

Политика замещения страниц

В идеале набор страниц, которые постоянно используются программой (так называемое **рабочее множество**), можно хранить в памяти, чтобы сократить количество ошибок из-за отсутствия страниц. Однако программисты обычно не знают, какие страницы находятся в рабочем множестве, поэтому операционная система периодически должна показывать это множество. Если программа обращается

к странице, которая отсутствует в основной памяти, ее нужно вызвать с диска. Однако чтобы освободить для нее место, на диск нужно отправить какую-нибудь другую страницу. Следовательно, нужен алгоритм для определения, какую именно страницу необходимо убрать из памяти.

Выбирать такую страницу просто наугад нельзя. Если, например, выбрать страницу, содержащую команду, выполнение которой вызвало ошибку, то при попытке вызвать следующую команду произойдет еще одна ошибка из-за отсутствия страницы. Большинство операционных систем стараются предсказать, какие из страниц в памяти наименее полезны в том смысле, что их отсутствие не повлияет сильно на ход программы. Иными словами, вместо того чтобы удалять страницу, которая скоро понадобится, постарайтесь выбрать такую страницу, которая не будет нужна долгое время.

По одному из алгоритмов удаляется та страница, которая использовалась наиболее давно, поскольку вероятность того, что она будет в текущем рабочем множестве, очень мала. Этот алгоритм называется **LRU (Least Recently Used — алгоритм удаления наиболее давно использовавшихся элементов)**. Хотя этот алгоритм работает достаточно хорошо, иногда возникают патологические ситуации. Ниже описана одна из них.

Представьте себе программу, выполняющую огромный цикл, который простирается на девять виртуальных страниц, а в физической памяти место есть только для восьми страниц. Когда программа перейдет к странице 7, в основной памяти будут находиться страницы с 0 по 7 (табл. 6.1). Затем совершается попытка вызвать команду из виртуальной страницы 8, что вызывает ошибку из-за отсутствия страницы. Нужно принять решение, какую страницу убрать. По алгоритму LRU будет выбрана виртуальная страница 0, поскольку она использовалась раньше всех. Виртуальная страница 0 удаляется, а нужная виртуальная страница помещается на ее место (табл. 6.2).

Таблица 6.1. Ситуация, в которой алгоритм LRU не действует (1)

Виртуальная страница 7
Виртуальная страница 6
Виртуальная страница 5
Виртуальная страница 4
Виртуальная страница 3
Виртуальная страница 2
Виртуальная страница 1
Виртуальная страница 0

Таблица 6.2. Ситуация, в которой алгоритм LRU не действует (2)

Виртуальная страница 7
Виртуальная страница 6
Виртуальная страница 5
Виртуальная страница 4
Виртуальная страница 3
Виртуальная страница 2
Виртуальная страница 1
Виртуальная страница 8

Таблица 6.3. Ситуация, в которой алгоритм LRU не действует (3)

Виртуальная страница 7
Виртуальная страница 6
Виртуальная страница 5
Виртуальная страница 4
Виртуальная страница 3
Виртуальная страница 2
Виртуальная страница 0
Виртуальная страница 8

После выполнения команд из виртуальной страницы 8 программа возвращается к началу цикла, то есть к виртуальной странице 0. Этот шаг вызывает еще одну ошибку из-за отсутствия страницы. Только что выброшенную виртуальную страницу 0 приходится вызывать обратно. По алгоритму LRU удаляется страница 1 (табл. 6.3). Через некоторое время программа пытается вызвать команду из виртуальной страницы 1, что опять вызывает ошибку. Затем вызывается страница 1 и удаляется страница 2 и т. д.

Очевидно, что в этой ситуации алгоритм LRU совершенно не работает (другие алгоритмы тоже не работают при сходных обстоятельствах). Однако если расширить размер рабочего множества, число ошибок из-за отсутствия страниц станет минимальным.

Можно применять и другой алгоритм — **FIFO (First-in First-out — первым поступил, первым выводится)**. FIFO удаляет ту страницу, которая раньше всех загружалась, независимо от того, когда в последний раз производилось обращение к этой странице. С каждым страничным кадром связан отдельный счетчик. Изначально все счетчики установлены на 0.

После каждой ошибки из-за отсутствия страниц счетчик каждой страницы, находящейся в памяти, увеличивается на 1, а счетчик только что вызванной страницы принимает значение 0. Когда нужно выбрать страницу для удаления, выбирается страница с самым большим значением счетчика. Поскольку она не загружалась в память очень давно, существует большая вероятность, что она больше не понадобится.

Если размер рабочего множества больше, чем число доступных страничных кадров, ни один алгоритм не дает хороших результатов, и ошибки из-за отсутствия страниц будут происходить часто. Если программа постоянно вызывает подобные ошибки, то говорят, что наблюдается **пробуксовка (thrashing)**. Думаю, не нужно объяснять, что пробуксовка очень нежелательна. Если программа использует большое виртуальное адресное пространство, но имеет небольшое медленно изменяющееся рабочее множество, которое помещается в основную память, ничего страшного не произойдет. Это утверждение имеет силу, даже если программа использует в сто раз больше слов виртуальной памяти, чем их содержится в физической основной памяти.

Если страница, которую нужно удалить, не менялась с тех пор, как ее считали (а это вполне вероятно, если страница содержит программу, а не данные), то необязательно записывать ее обратно на диск, поскольку точная копия там уже существует. Однако если эта страница изменилась, то копия на диске уже ей не соответствует, и ее нужно туда переписать.

Если мы научимся определять, изменялась ли страница или не изменялась, то сможем избежать ненужных переписываний на диск и сэкономим много времени. На многих машинах в контроллере управления памятью содержится один бит для каждой страницы, который равен 0 при загрузке страницы и принимает значение 1, когда микропрограмма или аппаратное обеспечение изменяют эту страницу. По этому биту операционная система определяет, менялась данная страница или нет и нужно ли ее перезаписывать на диск или нет.

Размер страниц и фрагментация

Если пользовательская программа и данные время от времени заполняют ровно целое число страниц, то когда они находятся в памяти, свободного места там не будет. С другой стороны, если они не заполняют ровно целое число страниц, на последней странице останется неиспользованное пространство. Например, если программа и данные занимают 26 000 байтов на машине с 4096 байтами на страницу, то первые 6 страниц будут заполнены целиком, что в сумме даст 6×4096=24 576 байтов, а последняя страница будет содержать 26 000–24576=1424 байта. Поскольку в каждой странице имеется пространство для 4096 байтов, 2672 байта останутся свободными. Всякий раз, когда седьмая страница присутствует в памяти, эти байты будут занимать место в основной памяти, но при этом не будут выполнять никакой функции. Эта проблема называется **внутренней фрагментацией** (поскольку неиспользованное пространство является внутренним по отношению к какой-то странице).

Если размер страницы составляет n байтов, то среднее неиспользованное пространство в последней странице программы будет n/2 байтов — ситуация подсказывает, что нужно использовать страницы небольшого размера, чтобы свести к минимуму количество неиспользованного пространства. С другой стороны, если страницы маленького размера, то потребуется много страниц и большая таблица страниц. Если таблица страниц сохраняется в аппаратном обеспечении, то для хранения большой таблицы страниц нужно много регистров, что повышает стоимость компьютера. Кроме того, при запуске и остановке программы на загрузку и сохранение этих регистров потребуется больше времени.

Более того, маленькие страницы снижают эффективность пропускной способности диска. Поскольку перед началом передачи данных с диска приходится ждать примерно 10 млс (поиск + время вращения), выгоднее совершать большие передачи. При скорости передачи данных 10 Мбайт в секунду передача 8 Кбайт добавляет всего 0,7 млс (по сравнению с передачей 1 Кбайт).

Однако у маленьких страниц есть свои преимущества. Если рабочее множество состоит из большого количества маленьких отделенных друг от друга областей виртуального адресного пространства, при маленьком размере страницы будет реже возникать пробуксовка (режим интенсивной подкачки), чем при большом. Рассмотрим матрицу A 10 000×10 000, которая хранится в последовательных 8-байтных словах (A[1,1], A[2,1], A[3,1] и т. д.). При такой записи элементы ряда 1 (A[1,1], A[1,2], A[1,3] и т. д.) будут начинаться на расстоянии 80 000 байтов друг от друга. Программа, выполняющая вычисление над элементами этого ряда, будет использовать 10 000 областей, каждая из которых отделена от следующей 79 992 байтами. Если бы размер страницы составлял 8 Кбайт, то для хранения всех страниц понадобилось бы 80 Мбайт.

При размере страницы в 1 Кбайт для хранения всех страниц потребуется всего 10 Мбайт ОЗУ. При размере памяти в 32 Мбайт и размере страницы в 8 Кбайт программа войдет в режим интенсивной подкачки, а при размере страницы в 1 Кбайт этого не произойдет.

Сегментация

До сих пор мы говорили об одномерной виртуальной памяти, в которой виртуальные адреса идут один за другим от 0 до какого-то максимального адреса. По многим причинам гораздо удобнее использовать два или несколько отдельных виртуальных адресных пространств. Например, компилятор может иметь несколько таблиц, которые создаются в процессе компиляции:

1. Таблица символов, которая содержит имена и атрибуты переменных.
2. Исходный текст, сохраняемый для распечатки.
3. Таблица, содержащая все использующиеся целочисленные константы и константы с плавающей точкой.
4. Дерево, содержащее синтаксический анализ программы.
5. Стек, используемый для вызова процедур в компиляторе.

Каждая из первых четырех таблиц постоянно растет в процессе компиляции. Последняя таблица растет и уменьшается совершенно непредсказуемо. В одномерной памяти эти пять таблиц пришлось бы разместить в виртуальном адресном пространстве в виде смежных областей, как показано на рис. 6.6.



Рис. 6.6. В одномерном адресном пространстве, в котором содержатся постоянно увеличивающиеся таблицы, одна из таблиц может «врезаться» в другую

Посмотрим, что произойдет, если программа содержит очень большое число переменных. Область адресного пространства, предназначенная для таблицы символов, может переполниться, даже если в других таблицах полно свободного места. Компилятор, конечно, может сообщить, что он не способен продолжать работу

из-за большого количества переменных, но можно без этого и обойтись, поскольку в других таблицах много свободного места.

Компилятор может забирать свободное пространство у одних таблиц и передавать его другим таблицам, но это похоже на управление оверлеями вручную — некоторое неудобство в лучшем случае и долгая скучная работа в худшем.

На самом деле нужно просто освободить программиста от расширения и сокращения таблиц, подобно тому как виртальная память исключает необходимость следить за разбиением программы на оверлеи.

Для этого нужно создать много абсолютно независимых адресных пространств, которые называются **сегментами**. Каждый сегмент состоит из линейной последовательности адресов от 0 до какого-либо максимума. Длина каждого сегмента может быть любой от 0 до некоторого допустимого максимального значения. Разные сегменты могут иметь разную длину (обычно так и бывает). Более того, длина сегмента может меняться во время выполнения программы. Длина стекового сегмента может увеличиваться всякий раз, когда что-либо помещается в стек, и уменьшаться, когда что-либо выталкивается из стека.

Так как каждый сегмент основывает отдельное адресное пространство, разные сегменты могут увеличиваться или уменьшаться независимо друг от друга и не влияя друг на друга. Если стеку в определенном сегменте понадобилось больше адресного пространства (чтобы стек мог увеличиться), он может получить его, поскольку в его адресном пространстве больше не во что «врезаться». Естественно, сегмент может переполниться, но это происходит редко, поскольку сегменты очень большие. Чтобы определить адрес в двухмерной памяти, программа должна выдавать номер сегмента и адрес внутри сегмента. На рис. 6.7 изображена сегментированная память.

Мы подчеркиваем, что сегмент является логическим элементом, о котором программист знает и который он использует. Сегмент может содержать процедуру, массив, стек или ряд скалярных переменных, но обычно в него входит только один тип данных.



Рис. 6.7. Сегментированная память позволяет увеличивать и уменьшать каждую таблицу независимо от других таблиц

Сегментированная память имеет и другие преимущества помимо упрощения работы со структурами данных, которые постоянно уменьшаются или увеличиваются. Если каждая процедура занимает отдельный сегмент, в котором первый

адрес — это адрес 0, то связывание процедур, которые компилируются отдельно, сильно упрощается. Когда все процедуры программы скомпилированы и связаны, при вызове процедуры из сегмента n для обращения к слову 0 будет использоваться адрес (n, 0).

Если процедура в сегменте n впоследствии изменяется и перекомпилируется, то другие процедуры менять не нужно (поскольку ни один начальный адрес не был изменен), даже если новая версия больше по размеру, чем старая. В одномерной памяти процедуры обычно располагаются друг за другом, и между ними нет никакого адресного пространства. Следовательно, изменение размера одной процедуры может повлиять на начальный адрес других процедур. Это, в свою очередь, требует изменения всех процедур, которые вызвают любую из этих процедур, чтобы попадать в новые начальные адреса. Если в программу включено много процедур, этот процесс будет слишком накладным.

Сегментация облегчает разделение общих процедур или данных между несколькими программами. Если компьютер содержит несколько программ, работающих параллельно (это может быть реальная или смоделированная параллельная обработка), и если все эти программы используют определенные процедуры, снабжать каждую программу отдельной копией расточительно для памяти. А если сделать каждую процедуру отдельным сегментом, их легко можно будет разделять между несколькими программами, что исключит необходимость создавать физические копии каждой разделяемой процедуры. В результате мы сэкономим память.

Разные сегменты могут иметь разные виды защиты. Например, сегмент с процедурой можно определить как «только для выполнения», запретив тем самым считывание из него и запись в него. Для массива с плавающей точк разрешается только чтение и запись, но не выполнение и т. д. Такая защита часто помогает обнаружить ошибки в программе.

Вы должны понимать, почему защита имеет смысл только в сегментированной памяти и не имеет никакого смысла в одномерной (линейной) памяти. Обычно в сегменте не могут содержаться и процедура и стек одновременно (только что-нибудь одно). Поскольку каждый сегмент включает в себя объект только одного типа, он может использовать защиту, подходящую для этого типа. Страничная организация памяти и сегментация сравниваются в табл. 6.4.

Таблица 6.4. Сравнение страничной организации памяти и сегментации

Свойства	Страничная организация памяти	Сегментация
Должен ли программист знать об этом?	Нет	Да
Сколько линейных адресных пространств имеется?	1	Много
Может ли виртуальное адресное пространство увеличивать размер памяти?	Да	Да
Легко ли управлять таблицами с изменяющимися размерами?	Нет	Да
Зачем была придумана такая технология?	Чтобы смоделировать память большого размера	Чтобы обеспечить несколько адресных пространств

Содержимое страницы в каком-то смысле случайно. Программист может ничего не знать о страничной организации памяти. В принципе можно поместить несколько битов в каждый элемент таблицы страниц для доступа к нему, но чтобы использовать эту особенность, программисту придется следить за границами страницы в адресном пространстве. Дело в том, что страничное разбиение памяти было придумано для устранения подобных трудностей. Поскольку у пользователя создается иллюзия, что все сегменты постоянно находятся в основной памяти, к ним можно обращаться и не следить при этом за оверлеями.

Как реализуется сегментация

Сегментацию можно реализовать одним из двух способов. Это подкачка и разбиение на страницы. При первом подходе некоторый набор сегментов находится в памяти в данный момент. Если происходит обращение к сегменту, которого нет в данный момент в памяти, этот сегмент переносится в память. Если для него нет места в памяти, один или несколько сегментов нужно сначала записать на диск (если копия уже не находится там; в этом случае соответствующая копия просто удаляется из памяти). В каком-то смысле подкачка сегментов очень похожа на вызов страниц по требованию: сегменты загружаются и удаляются только в случае необходимости.

Однако сегментация существенно отличается от разбиения на страницы в следующем: размер страниц фиксирован, а размер сегментов — нет. На рис. 6.8, *а* показан пример физической памяти, в которой изначально содержится 5 сегментов. Посмотрим, что происходит, если сегмент 1 удаляется, а сегмент 7, который меньше по размеру, помещается на его место. В результате получится конфигурация, изображенная на рис. 6.8, *б*. Между сегментом 7 и сегментом 2 находится неиспользованная область («дырка»). Затем сегмент 4 меняется на сегмент 5 (рис. 6.8, *в*), а сегмент 3 замещается сегментом 6 (рис. 6.8, *г*). Через некоторое время память разделится на ряд областей, одни из которых будут содержать сегменты, а другие — неиспользованные области. Это называется **внешней фрагментацией** (поскольку неиспользованное пространство попадает не в сегменты, а в «дырки» между ними, то есть процесс происходит вне сегментов). Иногда внешнюю фрагментацию называют **поклеточной разбивкой**.

Посмотрите, что произойдет, если программа обратится к сегменту 3 в тот момент, когда память подвергается внешней фрагментации (рис. 6.8, *г*). Общее пространство «дырок» составляет 10 Кбайт, а это больше, чем нужно для сегмента 3, но так как это пространство разбито на маленькие кусочки, сегмент 3 туда загрузить нельзя. Вместо этого приходится сначала удалять другой сегмент.

Чтобы избежать такой ситуации, нужно сделать следующее. Каждый раз, когда появляется «дырка», нужно перемещать сегменты, следующие за «дыркой», ближе к адресу 0, устраняя таким образом эту «дырку» и оставляя большую «дырку» в конце. Есть и другой способ. Можно подождать, пока внешняя фрагментация не примет серьезный оборот (когда на долю «дырок» приходится больше определенного процента от всего объема памяти), и только после этого совершить уплотнение. На рис. 6.8, *д* показано, как память будет выглядеть после уплотнения. Цель уплотнения памяти — собрать все маленькие «дырки» в одну большую «дырку», в которую можно поместить один или несколько сегментов. Недостаток

уплотнения состоит в том, что на этот процесс тратится некоторое количество времени. Совершать уплотнение после появления каждой дырки невыгодно.

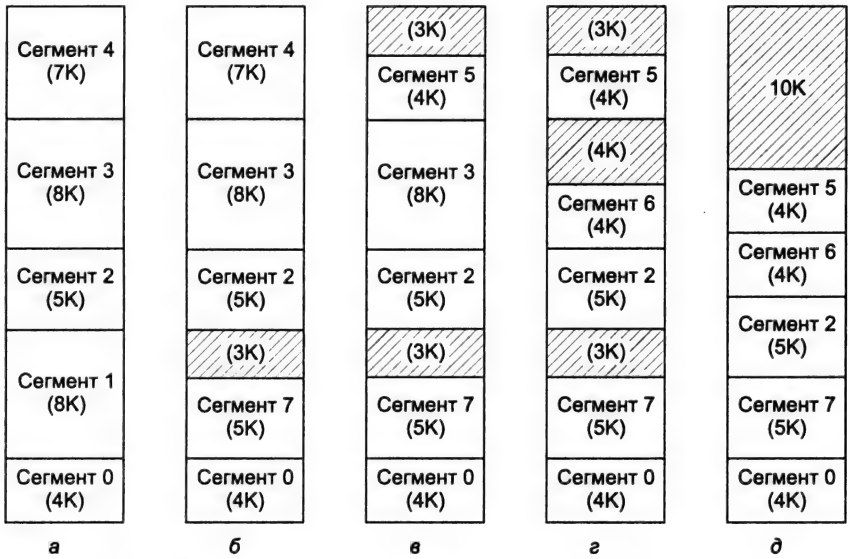


Рис. 6.8. Динамика внешней фрагментации (*а, б, в, г*); удаление внешней сегментации путем уплотнения (*д*)

Если на уплотнение памяти требуется слишком много времени, нужен специальный агоритм для определения, какую именно «дырку» лучше использовать для определенного сегмента. Для этого требуется список адресов и размеров всех «дырок». Популярный алгоритм **оптимальной подгонки** выбирает самую маленькую «дырку», в которую помещается нужный сегмент. Цель этого алгоритма — соотнести «дырки» и сегменты, чтобы избежать «отламывания» куска большой «дырки», который может понадобиться позже для большого сегмента.

Другой популярный алгоритм по кругу просматривает список «дырок» и выбирает первую «дырку», которая по размеру подходит для данного сегмента. Естественно, это занимает меньше времени, чем проверка всего списка, чтобы найти оптимальную «дырку». Удивительно, но последний алгоритм гораздо лучше, чем алгоритм оптимальной подгонки, с точки зрения общей производительности, поскольку оптимальная подгонка порождает очень много маленьких неиспользованных дырок [74].

Оба алгоритма сокращают средний размер «дырки». Всякий раз, когда сегмент помещается в «дырку», которая больше, чем этот сегмент, что бывает практически всегда (точные попадания очень редки), «дырка» делится на две части. Одну часть занимает сегмент, а вторая часть — это новая «дырка». Новая «дырка» всегда меньше, чем старая. Без воссоздания больших «дырок» из маленьких оба алгоритма в конечном итоге будут наполнять память маленькими неиспользованными «дырками».

Опишем один из таких процессов. Всякий раз, когда сегмент удаляется из памяти, а одна или обе соседние области этого сегмента — «дырки», а не сегменты, смежные неиспользованные пространства можно слить в одну большую «дырку». Если из рис. 6.8, *г* удалить сегмент 5, то две соседние «дырки» и 4 K, которые использовались данным сегментом, будут слиты в одну «дырку» в 11 K.

В начале этого раздела мы говорили, что реализовать сегментацию можно двумя способами: подкачкой и разбиением на страницы. До сих пор речь шла о подкачке. При таком подходе по необходимости между памятью и диском перемещаются целые сегменты.

Второй способ реализации — разделить каждый сегмент на страницы фиксированного размера и вызывать их по требованию. В этом случае одни страницы сегмента могут находиться в памяти, а другие — на диске. Чтобы разбить сегмент на страницы, для каждого сегмента нужна отдельная таблица страниц. Поскольку сегмент представляет собой линейное адресное пространство, все средства разбиения на страницы, которые мы до сих пор рассматривали, применимы к любому сегменту. Единственное различие состоит в том, что каждый сегмент получает отдельную таблицу страниц.

MULTICS (Multiplexed Information and Computing Service — служба общей информации и вычислений) — это древняя операционная система, которая совмещала сегментацию с разбиением на страницы. Она была разработана Массачусетским технологическим институтом совместно с компаниями Bell Labs и General Electric [28, 106]. Адреса в MULTICS состоят из двух частей: номера сегмента и адреса внутри сегмента. Для каждого процесса существовал сегмент дескриптора, который содержал дескриптор для каждого сегмента. Когда аппаратное обеспечение получало виртуальный адрес, номер сегмента использовался в качестве индекса в сегмент дескриптора для нахождения дескриптора нужного сегмента (рис. 6.9). Дескриптор указывал на таблицу страниц, что позволяло разбивать на страницы каждый сегмент обычным способом. Для повышения производительности недавно используемые комбинации сегмента и страницы помещались в ассоциативную память из 16 элементов. Операционная система MULTICS уже давно не применяется, но виртуальная память всех процессоров Intel, начиная с 386-го, очень похожа на эту систему.

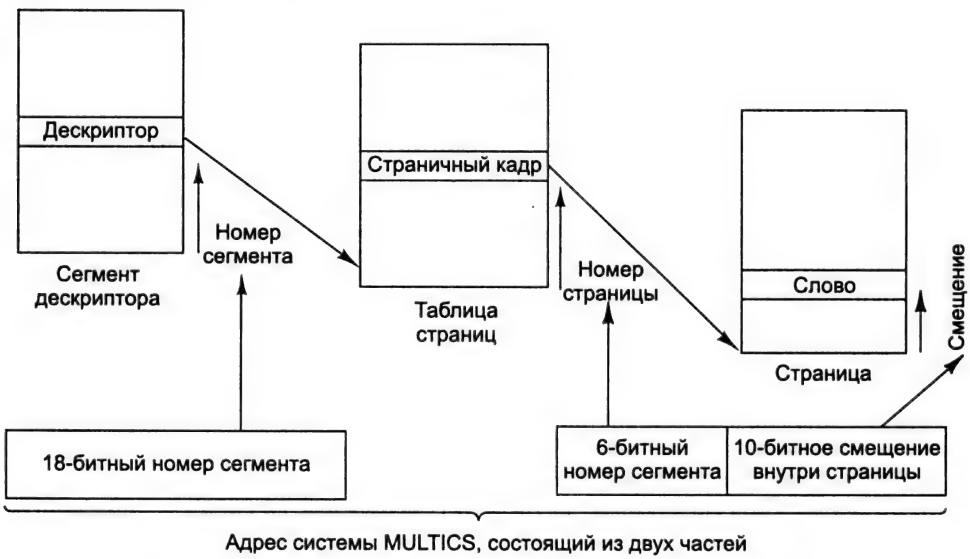


Рис. 6.9. Превращение адреса системы MULTICS, состоящего из двух частей, в адрес основной памяти

Виртуальная память в процессоре Pentium II

Pentium II имеет сложную систему виртуальной памяти, которая поддерживает вызов страниц по требованию, чистую сегментацию и сегментацию с разбиением на страницы. Виртуальная память состоит из двух таблиц: **LDT (Local Descriptor Table — локальная таблица дескрипторов)** и **GDT (Global Descriptor Table — глобальная таблица дескрипторов)**. Каждая программа имеет свою собственную локальную таблицу дескрипторов, а единственная глобальная таблица дескрипторов разделяется всеми программами компьютера. Локальная таблица дескрипторов LDT описывает локальные сегменты каждой программы (ее код, данные, стек и т. д.), а глобальная таблица дескрипторов GDT описывает системные сегменты, в том числе саму операционную систему.

Как мы уже говорили в главе 5, чтобы получить доступ к сегменту, Pentium II сначала загружает селектор для сегмента в один из сегментных регистров. Во время выполнения программы регистр CS содержит селектор для сегмента кода, DS содержит селектор для сегмента данных и т. д. Каждый селектор представляет собой 16-битное число (рис. 6.10).

Один из битов селектора показывает, является сегмент локальным или глобальным (то есть в какой из двух таблиц он находится: в локальной таблице дескрипторов или в глобальной таблице дескрипторов). Еще 13 битов определяют номер элемента в локальной или глобальной таблице дескрипторов, поэтому объем каждой из этих таблиц ограничен до 8 K (2^{13}) сегментных дескрипторов. Оставшиеся два бита связаны с защитой. Мы опишем их позже.

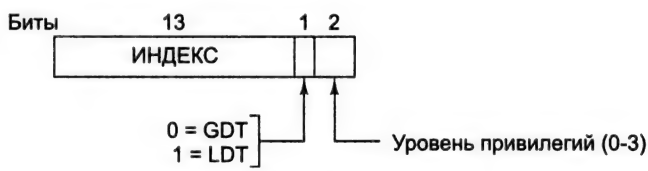


Рис. 6.10. Селектор в машине Pentium II

Дескриптор 0 недействителен и вызывает ловушку. Его можно загрузить в регистр сегмента, чтобы показать, что регистр сегмента в данный момент недоступен, но если попытаться использовать дескриптор 0, он вызовет ловушку.

Когда селектор загружается в сегментный регистр, соответствующий дескриптор вызывается из локальной таблицы дескрипторов или из глобальной таблицы дескрипторов и сохраняется во внутренних регистрах контроллера управления памятью, поэтому к нему можно быстро получить доступ. Дескриптор состоит из 8 байтов. Сюда входит базовый адрес сегмента, его размер и другая информация (рис. 6.11).

Формат селектора был выбран таким образом, чтобы упростить нахождение дескриптора. Сначала на основе бита 2 в селекторе выбирается локальная таблица дескрипторов LDT или глобальная таблица дескрипторов GDT. Затем селектор копируется во временный регистр контроллера управления памятью, а три младших бита принимают значение 0, в результате 13-битное число селектора умножается на 8. Наконец, к этому прибавляется адрес из локальной таблицы дескрипто-

ров или из глобальной таблицы дескрипторов (который хранится во внутренних регистрах контроллера управления памятью), и в результате получается указатель на дескриптор. Например, селектор 72 обращается к элементу 9 в глобальной таблице дескрипторов, который находится в ячейке с адресом GDT+72.

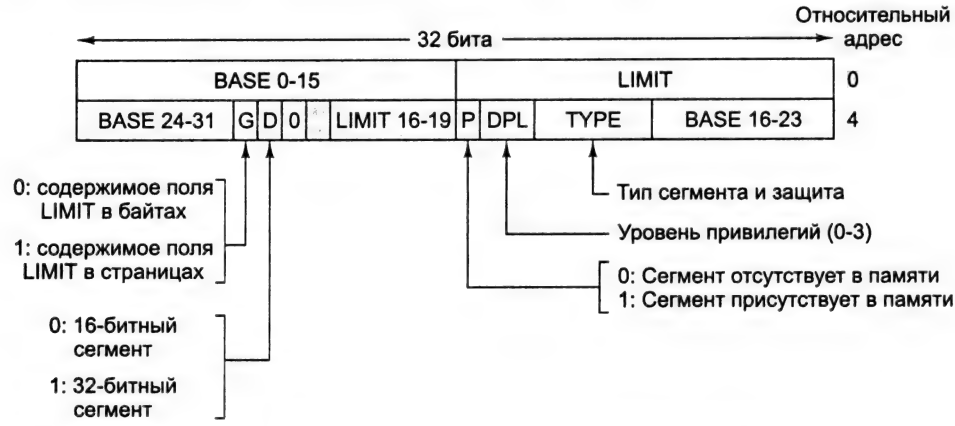


Рис. 6.11. Дескриптор сегмента кода в процессоре Pentium II. Сегменты данных практически ничем не различаются

Давайте проследим, каким образом пара (селектор, смещение) превращается в физический адрес. Как только аппаратное обеспечение определяет, какой именно регистр сегмента используется, оно может найти полный дескриптор, соответствующий данному селектору во внутренних регистрах. Если такого сегмента не существует (селектор 0) или в данный момент он не находится в памяти (P=0), вызывается системное прерывание (ловушка). В первом случае — это ошибка программирования; второй случай требует, чтобы операционная система вызвала нужный сегмент.

Затем аппаратное обеспечение проверяет, не выходит ли смещение за пределы сегмента. Если выходит, то снова происходит ловушка. По логике вещей в дескрипторе должно быть 32-битное поле для определения размера сегмента, но там в наличии имеется всего 20 битов, поэтому в данном случае используется совершенно другая схема. Если поле G (Granularity — степень детализации) равно 0, то поле LIMIT дает точный размер сегмента (до 1 Мбайт). Если поле G равно 1, то поле LIMIT указывает размер сегмента в страницах, а не в байтах. Размер страницы в компьютере Pentium II никогда не бывает меньше 4 Кбайт, поэтому 20 битов достаточно для сегментов до 2^{32} байтов.

Если сегмент находится в памяти, а смещение не выходит за границу сегмента, Pentium II прибавляет 32-битное поле BASE в дескрипторе к смещению, в результате чего получается **линейный адрес** (рис. 6.12). Поле BASE разбивается на три части и разносится по дескриптору, чтобы обеспечить совместимость с процессором 80286, в котором размер BASE составляет всего 24 бита. Поэтому каждый сегмент может начинаться с произвольного места в 32-битном адресном пространстве.

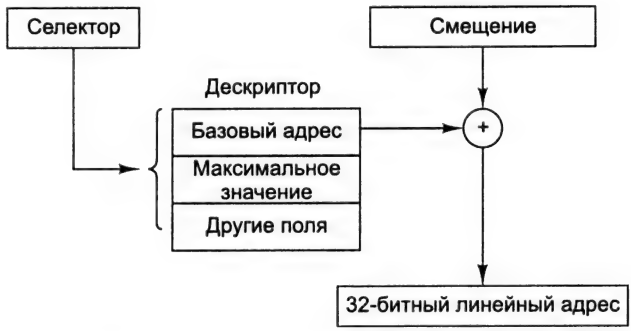


Рис. 6.12. Преобразование пары (селектор, смещение) в линейный адрес

Если разбиение на страницы блокировано (это определяется по биту в регистре глобального управления), линейный адрес интерпретируется как физический адрес и отправляется в память для чтения или записи. Таким образом, при блокировке разбиения на страницы мы имеем чистую схему сегментации, где каждый базовый адрес сегмента дан в его дескрипторе. Допускается перекрытие сегментов, поскольку было бы слишком утомительно тратить много времени на проверку, чтобы все сегменты были непересекающимися.

С другой стороны, если разбиение на страницы разрешено, линейный адрес интерпретируется как виртуальный адрес и отображается на физический адрес с использованием таблиц страниц, почти как в наших примерах. Единственная сложность состоит в том, что при 32-битном виртуальном адресе и страницах на 4 K сегмент может содержать 1 миллион страниц. Поэтому, чтобы сократить размер таблицы страниц для маленьких сегментов, применяется двухуровневое отображение.

Каждая работающая программа имеет специальную **таблицу страниц**, которая состоит из 1024 32-битных элементов. Ее адрес указывается глобальным регистром. Каждый элемент в этой таблице указывает на таблицу страниц, которая также содержит 1024 32-битных элементов. Элементы таблицы страниц указывают на страничные кадры. Схема изображена на рис. 6.13.

На рис. 6.13, *а* мы видим линейный адрес, разбитый на три поля: DIR, PAGE и OFF. Поле DIR используется в качестве индекса в директории страниц для нахождения указателя на нужную таблицу страниц. Поле PAGE используется в качестве индекса в таблице страниц для нахождения физического адреса страничного кадра. Наконец, поле OFF прибавляется к адресу страничного кадра, и получается физический адрес нужного байта или слова.

Размер каждого элемента таблицы страниц — 32 бита, 20 из которых содержат номер страничного кадра. Оставшиеся биты включают бит доступа и бит изменения, которые устанавливаются аппаратным обеспечением для помощи операционной системе, биты защиты и некоторые другие биты.

В каждой таблице страниц содержатся элементы для 1024 страничных кадров по 4 K каждый, поэтому одна таблица страниц работает с 4 Мбайт памяти. Сегмент короче 4 Мбайт будет иметь директорию страниц с одним элементом (указателем на его единственную таблицу страниц). Таким образом, непроизводительные издержки для коротких сегментов составляют всего две страницы, а не миллион страниц, как было бы в одноуровневой таблице страниц.

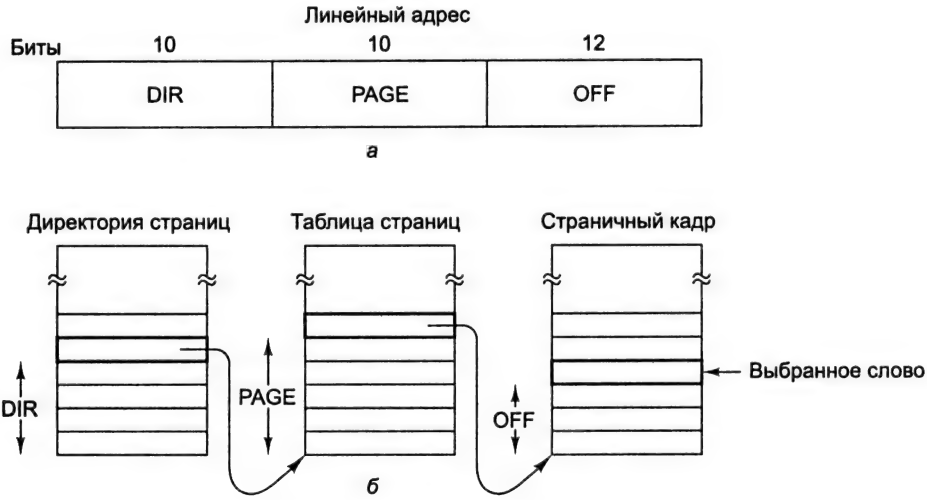


Рис. 6.13. Отображение линейного адреса на физический адрес

Чтобы избежать повторных обращений к памяти, устройство управления памятью Pentium II содержит специальную аппаратную поддержку для поиска недавно использовавшихся комбинаций DIR-PAGE и отображения их на физический адрес соответствующего страничного кадра. Шаги, показанные на рис. 6.13, выполняются только в том случае, если текущая комбинация не использовалась недавно.

При применении разбиения на страницы значение поля BASE в дескрипторе вполне может быть равно 0. Единственное, для чего нужно поле BASE, — это мотивировать небольшое смещение и использовать элемент в середине директории страниц, а не в начале. Поле BASE включено в дескриптор только для осуществления чистой сегментации (без разбиения на страницы), а также для обратной совместимости со старым процессором 80286, в котором не было разбиения на страницы.

Отметим, что если конкретное приложение не нуждается в сегментации и довольствуется единым 32-битным адресным пространством со страничной организацией, этого легко достичь. Все сегментные регистры могут быть заполнены одним и тем же селектором, дескриптор которого содержит поле BASE, равное 0, и поле LIMIT, установленное на максимальное значение. Смещение команды будет тогда линейным адресом с единственным адресным пространством — по сути, традиционное разбиение на страницы.

В завершение стоит сказать несколько слов о защите, поскольку это имеет непосредственное отношение к виртуальной памяти. Pentium II поддерживает 4 уровня защиты, где уровень 0 — самый привилегированный, а уровень 3 — наименее привилегированный. Они показаны на рис. 6.14. В каждый момент работающая программа находится на определенном уровне, указывающемся 2-битным полем в **PSW (Program Status Word — слово состояния программы)** — регистре аппаратного обеспечения, который содержит коды условия и другие биты состояния. Более того, каждый сегмент в системе также принадлежит к определенному уровню.

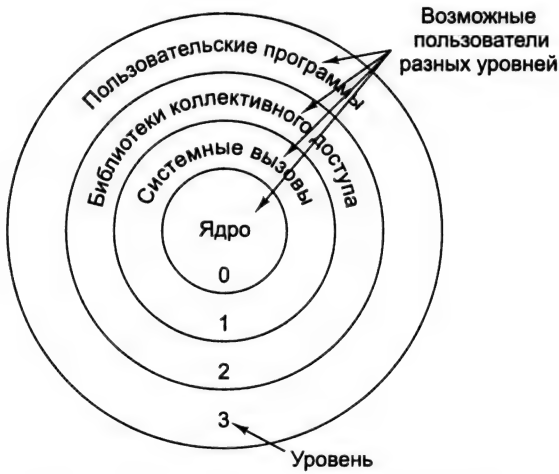


Рис. 6.14. Уровни защиты процессора Pentium II

Пока программа использует сегменты только своего собственного уровня, все идет нормально. Доступ к данным более высокого уровня разрешается. Доступ к данным более низкого уровня запрещен[1]: в этом случае происходит системное прерывание (ловушка). Допустим вызов процедур как более высокого, так и более низкого уровня, но при этом нужно вести строгий контроль. Для вызова процедуры из другого уровня команда CALL должна содержать селектор вместо адреса. Этот селектор обозначает дескриптор, который выдает адрес нужной процедуры. Таким образом, невозможно совершить переход в середину произвольного сегмента на другом уровне. Могут использоваться только официальные точки входа.

Рассмотрим рис. 6.14. На уровне 0 мы видим ядро операционной системы, которая контролирует процесс ввода-вывода, работу памяти и т. п. На уровне 1 находится обработчик системных вызовов. Пользовательские программы могут вызывать процедуры из этого уровня, но только строго определенные процедуры. Уровень 2 содержит библиотечные процедуры, которые могут разделяться несколькими работающими программами. Пользовательские программы могут вызывать эти процедуры, но не могут изменять их. На уровне 3 работают пользовательские программы, которые имеют самую низкую степень защиты. Система защиты Pentium II, как и схема управления памятью, в целом основана на идеях системы MULTICS.

Ловушки и прерывания используют механизм, сходный с описанным выше. Они тоже обращаются к дескрипторам, а не к абсолютным адресам, а эти дескрипторы указывают на процедуры, которые нужно выполнить. Поле FIELD на рис. 6.11 служит для различения сегментов кода, сегментов данных и различных типов логических элементов.

---

[1] Эти слова следует понимать следующим образом: автор называет более высоким уровнем те сегменты данных, у которых значение уровня привилегий больше, хотя на самом деле все обстоит иначе. Самым высоким уровнем привилегий считаются сегменты, имеющие уровень привилегий, равный 0, а самый низкий уровень привилегий имеют сегменты со значением этого уровня, равным 3. — *Примеч. научн. ред.*

Виртуальная память UltraSPARC II

UltraSPARC II — это 64-разрядная машина, которая поддерживает виртуальную память со страничной организацией и с 64-битными виртуальными адресами. Тем не менее по разным причинам программы не могут использовать полное 64-битное виртуальное адресное пространство. Поддерживается только 64 бита, поэтому программы не могут превышать $1{,}8 \times 10^{13}$ байтов. Допустимая виртуальная память делится на 2 зоны по 2^{43} байтов каждая, одна из которых находится в верхней части виртуального адресного пространства, а другая — в нижней. Между ними находится «дырка», содержащая адреса, которые не используются. Попытка использовать их вызовет ошибку из-за отсутствия страницы.

Максимальная физическая память компьютера UltraSPARC II составляет 2^{41} байтов (2200 Гбайт). Поддерживается 4 размера страниц: 8 Кбайт, 64 Кбайт, 512 Кбайт и 4 Мбайт. Отображения этих четырех размеров показаны на рис. 6.15.

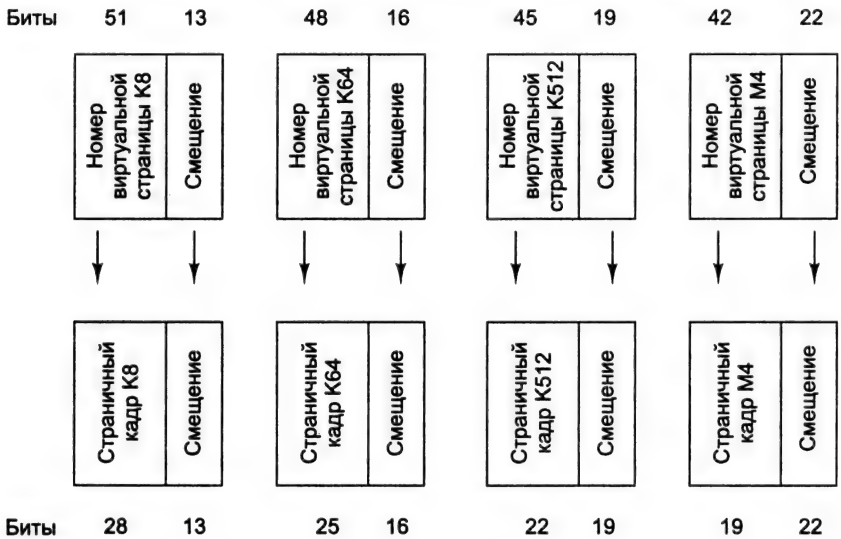


Рис. 6.15. Отображения виртуальных адресов в физические в машине UltraSPARC II

Из-за огромного виртуального адресного пространства обычная таблица страниц (как в Pentium II) не будет практичной. В UltraSPARC II применяется совершенно другой подход. Устройство управления памятью содержит таблицу, так называемый **TLB (Translation Lookaside Buffer — буфер быстрого преобразования адреса)**. Эта таблица отображает номера виртуальных страниц в номера физических страничных кадров. Для страниц размером в 8 К существует 2^{31} номеров виртуальных страниц, то есть более двух миллиардов. Естественно, не все они могут быть отображены.

Поэтому TLB содержит только номера самых последних используемых виртуальных страниц. Страницы команд и страницы данных рассматриваются отдельно. Для каждой из этих категорий в TLB включены номера 64 последних виртуальных страниц. Каждый элемент этого буфера включает номер виртуальной страницы и соответствующий номер физического страничного кадра. Когда номер процесса

вызывает его **контекст**, виртуальный адрес в этом контексте передается в контроллер управления памятью, то он с помощью специальной схемы сравнивает номер виртуальной страницы со всеми элементами буфера быстрого преобразования адреса TLB для данного контекста одновременно. Если обнаруживается совпадение, номер страничного кадра в этом элементе буфера соединяется со смещением, взятым из виртуального адреса, чтобы получить 41-битный физический адрес и обработать некоторые флаги (например, биты защиты). Буфер быстрого преобразования адресаTLB изображен на рис. 6.16, *а*.

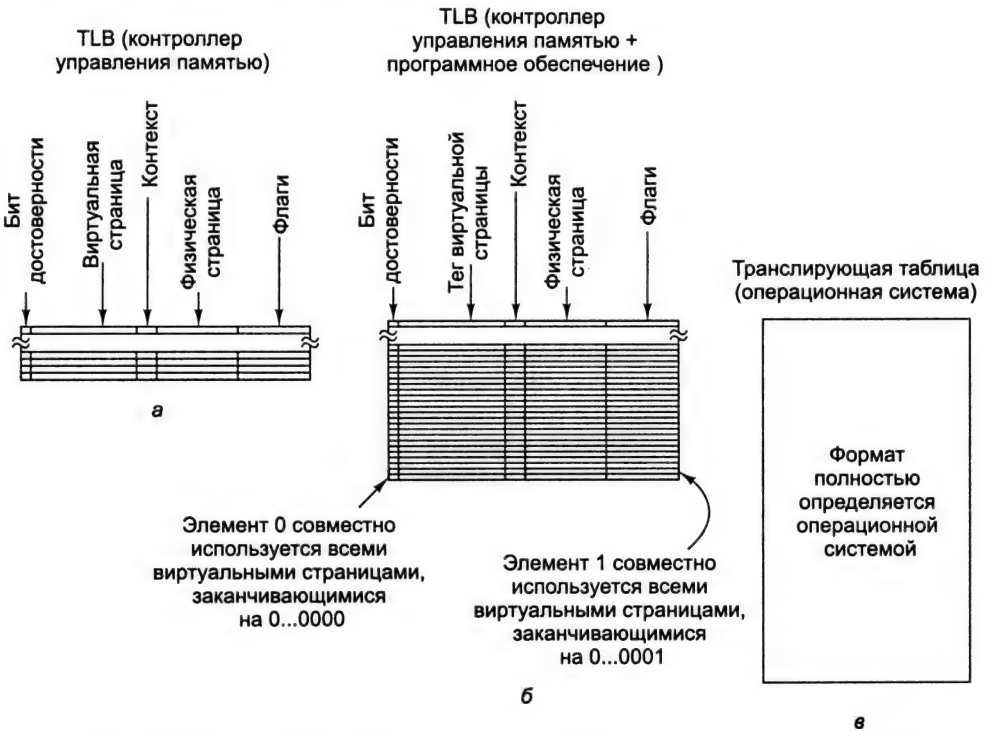


Рис. 6.16. Структуры данных, использующиеся для трансляции виртуального адреса на UltraSPARC II: буфер быстрого преобразования адреса TLB (*а*); буфер хранения преобразований (*б*); транслирующая таблица (*в*)

Если совпадение не обнаружилось, происходит **промах в TLB**, который вызывает ловушку в операционной системе. Обрабатывать ошибку должна сама операционная система. Отметим, что данный промах отличается от ошибки из-за отсутствия страницы. Промах буфера TLB может произойти, даже если нужная страница присутствует в памяти. Теоретически операционная система может сама загрузить новый элемент этого буфера для нужной виртуальной страницы. Однако для ускорения данной операции к этой работе подключается аппаратное обеспечение, если программное обеспечение взаимодействует с ним.

Операционная система должна сохранять наиболее часто используемые элементы буфера TLB в таблице под названием **буфер хранения преобразований**

(TSB — translation storage buffer). Эта таблица построена как кэш-память прямого отображения виртуальных страниц. Каждый 16-байтный элемент данной таблицы указывает на одну виртуальную страницу и содержит бит достоверности, номер контекста, тег виртуального адреса, номер физической страницы и несколько флаговых битов. Если размер кэш-памяти составляет, скажем, 8192 элемента, тогда все виртуальные страницы, у которых младшие 13 битов отображаются в 0000000000000, будут претендовать на элемент 0 в данной таблице. Точно так же все виртуальные страницы, у которых младшие биты отображаются в 0000000000001, претендуют на элемент 1 в этой таблице, как показано на рис. 6.16, *б*. Размер таблицы определяется программным обеспечением и передается в контроллер управления памятью через специальные регистры, доступные только для операционной системы.

При промахе буфера хранения преобразований операционная система проверяет, содержит ли соответствующий элемент буфера TLB нужную виртуальную страницу. Контроллер управления памятью вычисляет адрес этого элемента и помещает его в свой внутренний регистр, доступный для операционной системы. Если нужный элемент есть в таблице хранения преобразований, то какой-нибудь элемент удаляется из буфера TLB, а соответствующий элемент буфера хранения преобразований копируется туда. Аппаратное обеспечение с помощью алгоритма LRU выбирает, какой именно элемент нужно выкинуть.

Если нужной виртуальной страницы нет в кэш-памяти, операционная система использует другую таблицу для нахождения информации о странице, которая может находиться или не находиться в основной памяти. Таблица, которая применяется для этого поиска, называется **транслирующей таблицей**. Поскольку здесь аппаратное обеспечение не участвует в поиске элементов, операционная система может использовать любой формат. Например, она может хэшировать номер виртуальной страницы, разделив его на какое-либо число p, и использовать остаток для индексирования таблицы указателей, каждый из которых указывает на связный список виртуальных страниц, разделенных на p. Отметим, что эти элементы — не собственно страницы, а элементы таблицы TSB. Если поиск страницы в таблице трансляции привел к нахождению нужной страницы в памяти, то элемент TSB в кэш-памяти обновляется. Если в результате поиска обнаружилось, что нужной страницы нет в памяти, то происходит стандартная ошибка.

Сравним схемы разбиения на страницы в Pentium II и UltraSPARC II. Pentium II поддерживает чистую сегментацию, чистое разбиение на страницы и сегментацию в сочетании с разбиением на страницы. UltraSPARC II поддерживает только разбиение на страницы. Pentium II использует аппаратное обеспечение для перезагрузки элемента буфера TLB в случае промаха TLB. UltraSPARC II в случае такого промаха просто передает управление операционной системе.

Причина этого различия состоит в том, что Pentium II использует 32-битные сегменты, а такие маленькие сегменты (только 1 млн страниц) могут обрабатываться только с помощью страничных таблиц. Теоретически у Pentium II могли бы возникнуть проблемы, если бы программа использовала тысячи сегментов, но так как ни одна из версий Windows или UNIX не поддерживает более одного сегмента на процесс, никаких проблем не возникает. UltraSPARC II — 64-битная машина. Она может содержать до 2 млрд страниц, поэтому таблицы страниц не работают. В будущем все машины будут иметь 64-битные виртуальные адресные пространства, и схема UltraSPARC II станет нормой. Сравнение Pentium II, UltraSPARC II и четырех других схем виртуальной памяти можно найти в книге [66].

Виртуальная память и кэширование

На первый взгляд может показаться, что виртуальная память и кэширование никак не связаны, но на самом деле они сходны. При наличии виртуальной памяти вся программа хранится на диске и разбивается на страницы фиксированного размера. Некоторое подмножество этих страниц находится в основной памяти. Если программа главным образом использует страницы из основной памяти, то ошибки из-за отсутствия страницы будут встречаться редко и программа будет работать быстро. При кэшировании вся программа хранится в основной памяти и разбивается на блоки фиксированного размера. Некоторое подмножество этих блоков находится в кэш-памяти. Если программа главным образом использует блоки из кэш-памяти, то промахи кэш-памяти будут происходить редко и программа будет работать быстро. Как видим, виртуальная память и кэш-память идентичны, только работают они на разных уровнях иерархии.

Естественно, виртуальная память и кэш-память имеют некоторые различия. Промахи кэш-памяти обрабатываются аппаратным обеспечением, а ошибки из-за отсутствия страниц обрабатываются операционной системой. Блоки кэш-памяти обычно гораздо меньше страниц (например, 64 байта и 8 Кбайт). Кроме того, таблицы страниц индексируются по старшим битам виртуального адреса, а кэш-память индексируется по младшим битам адреса памяти. Тем не менее важно понимать, что различие здесь только в реализации.

Виртуальные команды ввода-вывода

Набор команд уровня архитектуры команд полностью отличается от набора команд микроархитектурного уровня. И сами операции, и форматы команд на этих двух уровнях различны. Наличие нескольких одинаковых команд случайно.

Набор команд уровня операционной системы содержит большую часть команд из уровня архитектуры команд, а также несколько новых очень важных команд. Некоторые ненужные команды в уровень операционной системы не включаются. Ввод-вывод — это одна из областей, в которых эти два уровня различаются очень сильно. Причина такого различия проста. Во-первых, пользователь, способный выполнять команды ввода-вывода уровня архитектуры команд, сможет считать конфиденциальную информацию, которая хранится где-нибудь в системе, и вообще будет представлять угрозу для самой системы. Во-вторых, обычные нормальные программисты не хотят осуществлять ввод-вывод на уровне команд, поскольку это слишком сложно и утомительно. Вместо этого для осуществления ввода-вывода нужно установить определенные поля и биты в ряде регистров устройств, затем подождать, пока операция закончится, и проверить, что произошло. Диски обычно содержат биты регистров устройств для обнаружения следующих ошибок.

1. Аппаратура диска не смогла выполнить позиционирование.
2. Несуществующий элемент памяти определен как буфер.
3. Процесс ввода-вывода с диска (на диск) начался до того, как закончился предыдущий.

4. Ошибка синхронизации при считывании.
5. Обращение к несуществующему диску.
6. Обращение к несуществующему цилиндру.
7. Обращение к несуществующему сектору.
8. Ошибка проверки записи после операции записи.

При наличии одной из этих ошибок устанавливается соответствующий бит в регистре устройства.

Файлы

Один из способов организации виртуального ввода-вывода — использование абстракции под названием **файл**. Файл состоит из последовательности байтов, записанных на устройство ввода-вывода. Если устройство ввода-вывода является устройством хранения информации (например, диск), то файл можно считать обратно. Если устройство не является устройством хранения информации (например, это принтер), то файл оттуда считать нельзя. На диске может храниться много файлов, в каждом из которых содержатся данные опредленного типа, например картинка, крупноформатная таблица или текст. Файлы имеют разную длину и обладают разными свойствами. Эта абстракция позволяет легко организовать виртуальный ввод-вывод.

Для операционной системы файл является просто последовательностью байтов, как мы и описали выше. Ввод-вывод файла осуществляется с помощью системных вызовов для открытия, чтения, записи и закрытия файлов. Перед тем как считывать файл, его нужно открыть. Процесс открытия файла позволяет операционной системе найти файл на диске и передать в память информацию, необходимую для доступа к этому файлу.

После открытия файла его можно считывать. Системный вызов для считывания должен иметь как минимум следующие параметры:

1. Указание, какой именно открытый файл нужно считывать.
2. Указатель на буфер в памяти, в который нужно поместить данные.
3. Число считываемых байтов.

Данный системный вызов помещает требующиеся данные в буфер. Обычно он возвращает число считанных байтов. Это число может быть меньше того числа, которое запрашивалось изначально (например, нельзя считать 2000 байтов из файла размером 1000 байт).

С каждым открытым файлом связан указатель, который сообщает, какой байт будет считываться следующим. После команды `read` указатель дополняется числом считанных байтов, поэтому последовательные команды `read` считывают последовательные блоки данных из файла. Обычно этот указатель можно установить на особое значение, чтобы программы могли получать доступ к любой части файла. Когда программа закончила считывание файла, она может закрыть его и сообщить операционной системе, что она больше не будет использовать этот файл. Операционная система сможет освободить пространство в таблице, в которой хранилась информация об этом файле.

В операционных системах для универсальных вычислительных машин файл представляет собой более сложную структуру. Здесь файл может быть последовательностью **логических записей**, каждая из которых имеет строго определенную структуру. Например, логическая запись может представлять собой структуру данных, состоящую из пяти элементов: двух строк символов, «Имя» и «Начальник», двух целых чисел «Отдел» и «Комната», и одного логического числа «Пол женский». Некоторые операционные системы различают файлы, в которых все элементы имеют одинаковую структуру, и файлы, содержащие разные типы данных.

Основная виртуальная команда ввода считывает следующую запись из нужного файла и помещает ее в основную память, начиная с определенного адреса, как показано на рис. 6.17. Чтобы выполнить эту операцию, виртуальная команда должна получить сведения о том, какой файл считывать и куда в памяти поместить запись. Часто существуют параметры для чтения некоторой записи, которые определяются или по ее месту в файле, или по ее ключу.

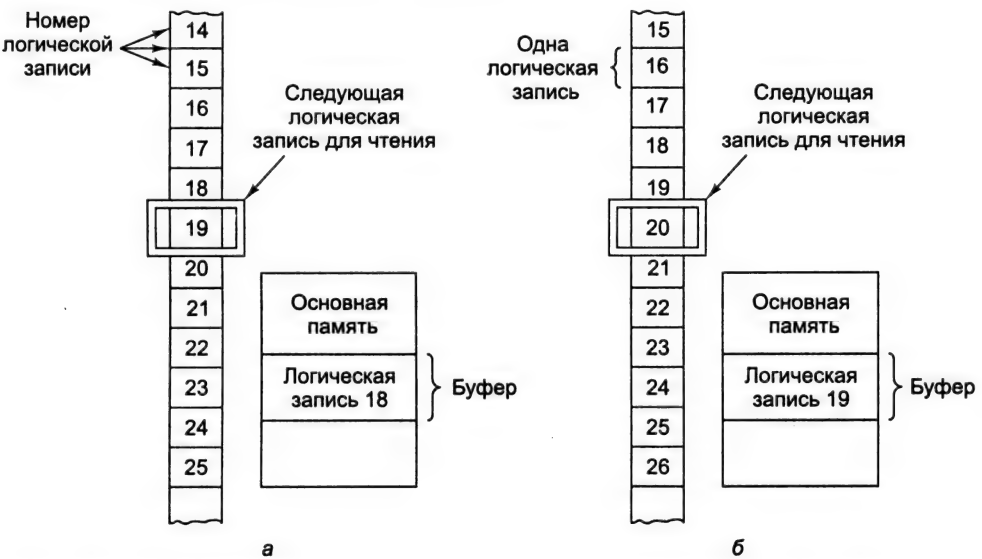


Рис. 6.17. Чтение файла, состоящего из логических записей: до чтения записи 19 (*а*); после чтения записи 19 (*б*)

Основная виртуальная команда вывода записывает логическую запись из памяти в файл. Последовательные команды `write` производят последовательные логические записи в файл.

Реализация виртуальных команд ввода-вывода

Чтобы понять, как реализуются виртуальные команды ввода-вывода, нужно изучить, как файлы организуются и хранятся. Основной вопрос здесь — распределение памяти. Единичным блоком может быть один сектор на диске, но чаще он состоит из нескольких последовательных секторов.

Еще одно фундаментальное свойство реализации системы файлов — хранится ли файл в последовательных блоках или нет. На рис. 6.18 изображен простой диск с одной поверхностью, состоящий из 5 дорожек по 12 секторов каждая. На рис. 6.18, *а* файл состоит из последовательных секторов. Последовательное расположение пяти блоков обычно применяется на компакт-дисках. На рис. 6.18, *б* файл занимает непоследовательные сектора. Такая схема является нормой для жестких дисков.

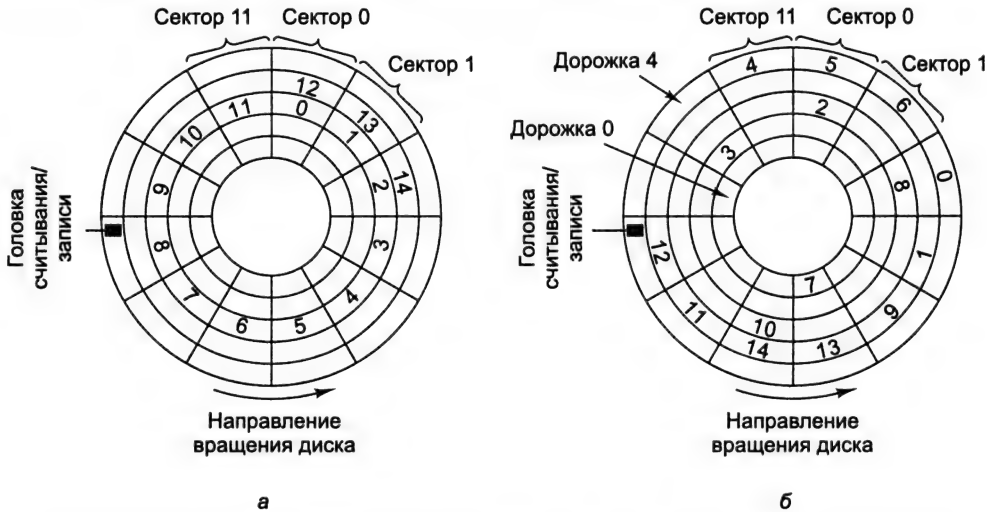


Рис. 6.18. Варианты расположения файла на диске: файл занимает последовательные сектора (*а*); файл занимает непоследовательные сектора (*б*)

Восприятие файла прикладным программистом сильно отличается от восприятия файла операционной системой. Программист воспринимает файл как линейную последовательность байтов или логических записей. Операционная система воспринимает файл как упорядоченную, хотя необязательно последовательную, совокупность единичных блоков на диске.

Для того чтобы операционная система могла доставить байт или логическую запись n из какого-то файла по требованию, она должна пользоваться каким-либо методом для определения местонахождения данных. Если файл расположен последовательно, операционная система должна знать только место начала файла, чтобы вычислить позицию нужного байта или логической записи.

Если файл расположен на диске непоследовательно, то невозможно только по начальной позиции файла вычислить позицию произвольного байта или логической записи в этом файле. Чтобы найти произвольный байт или логическую запись, нужна таблица (так называемый **индекс файла**), которая выдает единичные блоки и их физические адреса на диске. Индекс файла может быть организован либо в виде списка адресов блоков (такая схема используется в UNIX), либо в виде списка логических записей, для каждой из которых дается адрес на диске и смещение. Иногда каждая логическая запись имеет **ключ**, и программы могут обращаться к записи по этому ключу, а не по номеру логической записи. В последнем случае

каждый элемент таблицы должен содержать не только информацию о местонахождении записи на диске, но и ее ключ. Такая структура обычно применяется в универсальных вычислительных машинах.

Альтернативный метод нахождения блоков файла — организовать файл в виде связного списка. Каждый единичный блок содержит адрес следующего единичного блока. Для реализации этой схемы нужно в основной памяти иметь таблицу со всеми последующими адресами. Например, для диска с 64 К блоками операционная система может иметь в памяти таблицу из 64 К элементов, в каждом из которых дается индекс следующего единичного блока. Так, если файл занимает блоки 4, 52 и 19, то элемент 4 в таблице будет содержать число 52, элемент 52 будет содержать число 19, а элемент 19 будет содержать специальный код (например, 0 или −1), который указывает на конец файла. Так работают системы файлов в MS DOS, Windows 95 и Windows 98. Windows NT поддерживает эту систему файлов, но кроме этого имеет свою собственную систему файлов, которая больше похожа на UNIX.

До сих пор мы обсуждали как последовательно расположенные, так и непоследовательно расположенные на диске файлы, но мы еще не объяснили, зачем нужны эти два типа расположения. Последовательно расположенными файлами легко управлять, но если максимальный размер файла не известен заранее, эту технологию нельзя использовать. Если файл начинается с сектора j и разрастается в последовательные сектора, он может наткнуться на другой файл в секторе k, и ему не хватит места на расширение. Если файл располагается непоследовательно, то таких проблем не возникает, поскольку следующие блоки можно поместить в другое место на диске. Если диск содержит ряд увеличивающихся файлов, конечные размеры которых неизвестны, записать их в последовательные блоки на диске практически невозможно. Иногда можно переместить существующий файл, но это очень накладно.

С другой стороны, если максимальный размер всех файлов известен заранее (например, это имеет место, когда создается компакт-диск), программа может заранее определить серии секторов, точно равных по длине каждому файлу. Если файлы по 1200, 700, 2000 и 900 секторов нужно поместить на компакт-диск, они просто могут начинаться с секторов 0, 1200, 1900 и 3900 соответственно (оглавление здесь не учитывается). Найти любую часть любого файла легко, поскольку известен первый сектор файла.

Чтобы распределить пространство на диске для файла, операционная система должна следить, какие блоки доступны, а какие уже заняты другими файлами. При записи на компакт-диск вычисление производится один раз и навсегда, а на жестком диске файлы постоянно записываются и удаляются. Один из способов — сохранить список всех «дырок» (неиспользованных пространств), где «дырка» — это любое число смежных единичных блоков. Этот список называется **списком свободной памяти**. На рис. 6.19, *а* изображен список свободной памяти для диска с рис. 6.18, *б*.

Альтернативный подход — сохранить битовое отображение, один бит на единичный блок, как показано на рис. 6.19, *б*. Бит со значением 1 показывает, что данный блок уже занят, а бит со значением 0 показывает, что данный блок свободен.

Дорожка	Сектор	Количество секторов в дырке
0	0	5
0	6	6
1	0	10
1	11	1
2	1	1
2	3	3
2	7	5
3	0	3
3	9	3
4	3	8

а

	Сектор											
Дорожка	0	1	0	1	0	1	0	1	0	1	0	1
0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	1	0	0	0	0	0
3	0	0	0	1	1	1	1	1	1	0	0	0
4	1	1	1	0	0	0	0	0	0	0	0	1

б

Рис. 6.19. Два способа отслеживания свободных секторов: список свободной памяти (*а*); битовое отображение (*б*)

Первый подход позволяет легко находить дырку определенной длины. Однако у этого метода есть недостаток: по мере создания и уничтожения файлов длина списка будет меняться, а это нежелательно. Преимущество таблицы битов состоит в том, что она имеет постоянный размер. Кроме того, для изменения статуса единичного блока из свободного в занятый нужно поменять значение всего одного бита. Однако при таком подходе трудно найти блок данного размера. Оба метода требуют, чтобы при записи файла на диск или удалении файла с диска список размещения или таблица обновлялись.

Перед тем как закончить обсуждение вопроса о реализации системы файлов, нужно сказать пару слов о размере единичного блока. Здесь играют роль несколько факторов. Во-первых, время поиска и время, затрачиваемое на вращение диска, затормаживают доступ к диску. Если на нахождение начала блока тратится 10 млс, то гораздо выгоднее считать 8 Кбайт (это займет примерно 1 млс), чем 1 Кбайт (это займет примерно 0,125 миллисекунды), так как если считывать 8 Кбайт как 8 блоков по 1 Кбайт, нужно будет осуществлять поиск 8 раз. Для повышения производительности нужны большие единичные блоки.

Чем меньше размер единичного блока, тем больше их должно быть. Большое количество единичных блоков, в свою очередь, влечет за собой длинные индексы файлов и большие таблицы в памяти. Системе MS DOS пришлось перейти на многосекторные блоки по той причине, что дисковые адреса хранились в виде 16-битных чисел. Когда размер дисков стал превышать 64 К секторов, представить их можно было, только используя единичные блоки большего размера, поэтому число таких блоков не превышало 64 К. В первом выпуске системы Windows 95 возникла та же проблема, но в последующем выпуске уже использовались 32-битные числа. Система Windows 98 поддерживает оба размера.

Маленькие блоки тоже имеют свои преимущества. Дело в том, что файлы очень редко занимают ровно целое число единичных блоков. Следовательно, практически в каждом файле в последнем единичном блоке останется неиспользованное пространство. Если размер файла сильно превышает размер единичного блока, то в среднем неиспользованное пространство будет составлять половину блока. Чем

больше блок, тем больше остается неиспользованного пространства. Если средний размер файла намного меньше размера единичного блока, большая часть пространства на диске будет неиспользованной. Например, в системе MS DOS или в первой версии Windows 95 с разделом диска в 2 Гбайт размер единичного блока составляет 32 Кбайт, поэтому при записи на диск файла в 100 символов 32 668 байтов дискового пространства пропадут. С точки зрения распределения дискового пространства маленькие единичные блоки имеют преимущество над большими. В настоящее время самым важным фактором считается быстрота передачи данных, поэтому размер блоков постоянно увеличивается.

Команды управления директориями

Много лет назад программы и данные хранились на перфокартах. Поскольку размер программ увеличивался, а данных становилось все больше, такая форма хранения стала неудобной. Тогда возникла идея вместо перфокарт использовать для хранения программ и данных вспомогательную память (например, диск). Информация, доступная для компьютера без вмешательства человека, называется **неавтономной**. **Автономная** информация, напротив, требует вмешательства человека (например, нужно вставить компакт-диск).

Неавтономная информация хранится в файлах. Программы могут получить доступ к ней через программы ввода-вывода. Чтобы следить за информацией, записанной неавтономно, группировать ее в удобные блоки и защищать от незаконного использования, нужны дополнительные команды.

Обычно операционная система группирует неавтономные файлы в **директории**. На рис. 6.20 проиллюстрирован пример такой организации. Обеспечиваются системные вызовы, по крайней мере, для следующих функций:

1. Создание файла и введение его в директорию.
2. Стирание файла из директории.
3. Переименование файла.
4. Изменение статуса защиты файла.

Применяются различные схемы защиты. Например, владелец файлов может ввести секретный пароль для каждого файла. При попытке доступа к файлу программа должна выдавать пароль, который проверяется операционной системой. Доступ к файлу разрешается только в том случае, если пароль правильный. Для каждого файла можно создать список людей, программы которых могут получать доступ к данному файлу.

Все современные операционные системы позволяют хранить более одной директории. Каждая директория обычно сама представляет собой файл и как таковая может быть вставлена в другую директорию, в результате чего можно получить дерево директорий. Большое количество директорий особенно нужно программистам, которые работают над несколькими проектами. Они могут сгруппировать в одну директорию все файлы, связанные с одним проектом. Директории — это удобный способ делить файлы с членами своих рабочих групп.

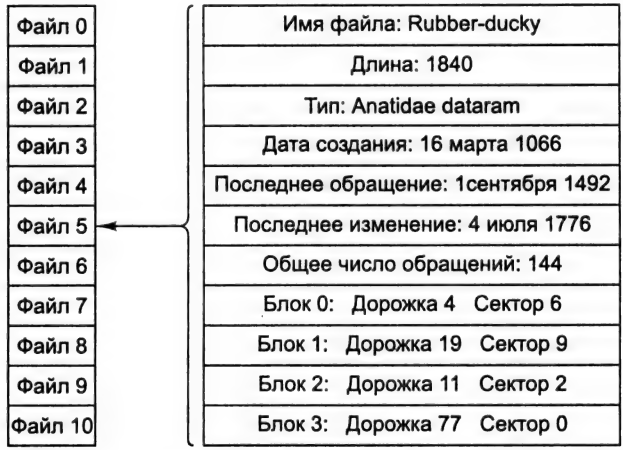


Рис. 6.20. Пользовательская директория (*а*); содержание одного из элементов директории (*б*)

Виртуальные команды для параллельной обработки

Некоторые вычисления удобно производить с помощью двух и более параллельных процессов (то есть как будто бы на разных процессорах). Другие вычисления можно поделить на части, которые затем выполняются параллельно, что сокращает общее время вычисления. Чтобы несколько процессов могли происходить параллельно, нужны специальные виртуальные команды. Мы будем их обсуждать в следующих разделах.

Интерес к параллельной обработке обусловлен и некоторыми законами физики. Согласно эйнштейновской теории относительности, скорость передачи электрических сигналов не может превышать скорость света, которая равна примерно 1фут/нс в вакууме, а в медном проводе или оптическом волокне — еще меньше. Важно учитывать этот предел при разработке компьютеров. Например, если процессору нужны данные из основной памяти, которые находятся на расстоянии 1 фута, потребуется по крайней мере 1 нс, чтобы запрос дошел до памяти, и еще 1 нс, чтобы ответ венулся к центральному процессору. Следовательно, чтобы компьютеры могли передавать сигналы быстрее, они (компьютеры) должны быть совершенно крошечными. Альтернативный способ увеличения скорости компьютера — создание машины с несколькими процессорами. Компьютер, содержащий 1000 процессоров с временем цикла в 1 нс, будет иметь такую же мощность, как процессор с временем цикла 0,001 нс, но первое осуществить гораздо проще и дешевле.

В компьютере с несколькими процессорами каждый из нескольких взаимодействующих процессов можно приписать к одному определенному процессору, чтобы выполнять несколько действий одновременно. Если в компьютере имеется только один процессор, эффект параллельной обработки можно смоделировать. Для этого процессы будут выполняться по очереди, каждый понемножку. Иными словами, процессор будет разделяться между несколькими процессами.

На рис. 6.21 показана разница между реальной параллельной обработкой, когда присутствует несколько физических процессоров, и смоделированной параллельной обработкой, когда имеется всего один физический процессор. Даже если параллельная обработка моделируется, удобно считать, что каждому процессу приписывается его собственный виртуальный процессор. При моделированной параллельной обработке возникают те же проблемы, что и при реальной параллельной обработке.

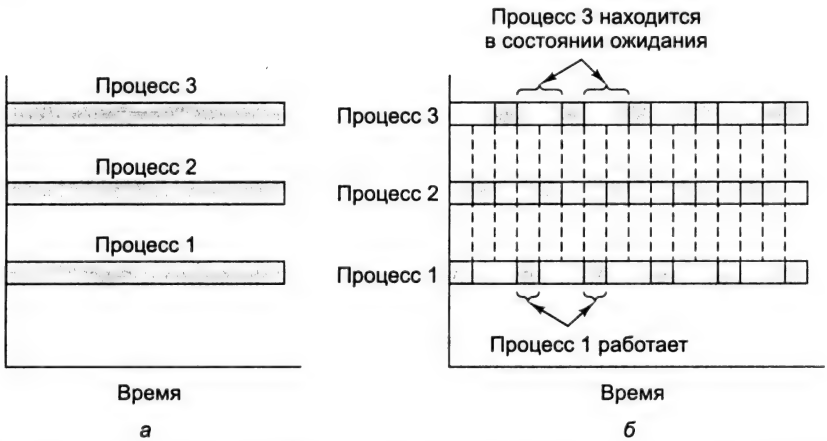


Рис. 6.21. Параллельная обработка с несколькими процессорами (*а*); моделирование параллельной обработки путем переключения одного процессора с одного процесса на другой (в данном случае всего три процесса) (*б*)

Формирование процесса

Программа должна работать как часть какого-либо процесса. Этот процесс, как и все другие процессы, характеризуется состоянием и адресным пространством, через которое можно получить доступ к программам и данным. Состояние включает как минимум счетчик команд, слово состояния программы, указатель стека и регистры общего назначения.

Большинство современных операционных систем позволяют формировать и прерывать процессы динамически. Для формирования нового процесса требуется системный вызов. Этот системный вызов может просто создать клон вызывающей программы или позволить исходному процессу определить начальное состояние нового процесса, то есть его программу, данные и начальный адрес.

В одних случаях исходный процесс сохраняет частичный или даже полный контроль над порожденным процессом. Виртуальные команды позволяют исходному процессу останавливать и снова запускать, проверять и завершать подчиненные процессы. В других случаях исходный процесс никак не контролирует порожденный процесс: после того как новый процесс сформирован, исходный процесс не может его остановить, запустить заново, проверить или завершить. Таким образом, эти два процесса работают независимо друг от друга.

Состояние гонок

Во многих случаях параллельные процессы должны взаимодействовать, и их работу нужно синхронизировать. В этом разделе мы рассмотрим синхронизацию и некоторые трудности, связанные с ней. Способы разрешения этих трудностей будут представлены в следующем разделе.

Рассмотрим два независимых процесса, процесс 1 и процесс 2, которые взаимодействуют через общий буфер в основной памяти. Для простоты мы будем называть процесс 1 **производителем** (producer), а процесс 2 — **потребителем** (consumer). Производитель выдает простые числа и помещает их в буфер по одному. Потребитель удаляет их из буфера по одному и печатает.

Эти два процесса работают параллельно с разной скоростью. Если производитель обнаруживает, что буфер заполнен, он переходит в режим ожидания, то есть временно приостанавливает операцию и ожидает сигнала от потребителя. Когда потребитель удаляет число из буфера, он посылает сигнал производителю, чтобы тот возобновил работу. Если потребитель обнаруживает, что буфер пуст, он приостанавливает работу. Когда производитель помещает число в пустой буфер, он посылает соответствующий сигнал потребителю.

В нашем примере для взаимодействия процессов мы будем использовать кольцевой буфер. Указатели *in* и *out* будут использоваться следующим образом: *in* указывает на следующее свободное слово (куда производитель поместит следующее простое число), а *out* указывает на следующее число, которое должен удалить потребитель.

Если *in*=*out*, буфер пуст, как показано на рис. 6.22, *а*. На рис. 6.22, *б* показана ситуация после того, как производитель породил несколько простых чисел. На рис. 6.22, *в* изображен буфер после того, как потребитель удалил из него несколько простых чисел для печати. На рис. 6.22, *г* — *е* представлены промежуточные стадии работы буфера. Буфер заполняется по кругу.

Если в буфер было отправлено слишком много чисел и он начал заполняться по второму кругу (снизу), а под *out* есть только одно слово (например, *in*=52, а *out*=53), буфер заполнится. Последнее слово не используется; если бы оно использовалось, то не было бы возможности сообщить, что именно значит равенство *in*=*out* — полный буфер или пустой буфер.

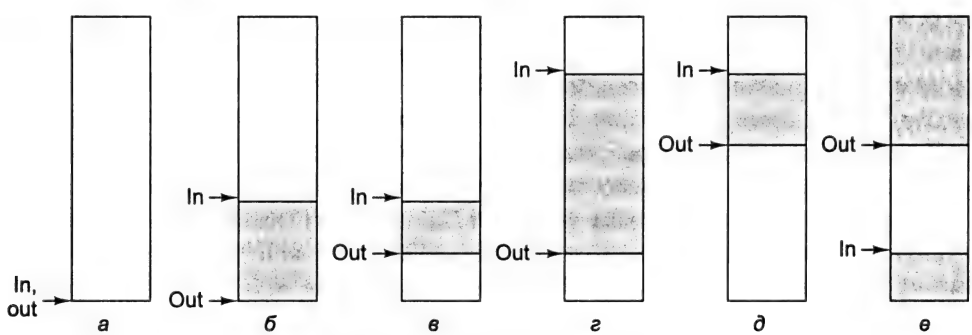


Рис. 6.22. Кольцевой буфер

В листинге 6.1 на языке Java показано решение задачи с производителем и потребителем.

Здесь используются три класса: *m*, *produser* и *consumer*. Класс *m* содержит некоторые константы, указатели буфера *in* и *out* и сам буфер, который в нашем примере вмещает 100 простых чисел (от buffer[0] до buffer[99]).

Для моделирования параллельных процессов в данном случае используются **потоки (threads)**. У нас есть класс *producer* и класс *consumer*, которым приписываются значения переменных *p* и *c* соответственно. Каждый из этих классов образуется из базового класса *Thread* с процедурой *run*. Класс *run* содержит код для *thread*. Когда вызывается процедура *start* для объекта, образованного из *Thread*, запускается новый поток.

Каждый поток похож на процесс. Единственным различием является то, что все потоки в пределах одной программы на языке Java работают в одном адресном пространстве. Это позволяет им разделять один общий буфер. Если в компьютере имеется два и более процессоров, каждый поток может выполняться на другом процессоре, поэтому в данном случае имеет место реальный параллелизм. Если компьютер содержит только один процессор, потоки разделяются во времени на одном процессоре. Мы будем продолжать называть производителя и потребителя процессами (поскольку нас в данный момент интересуют параллельные процессы), хотя Java поддерживает только параллельные потоки, а не реальные параллельные процессы.

Функция *next* позволяет увеличивать значения *in* и *out*, при этом не нужно каждый раз записывать код, чтобы проверить условие циклического возврата. Если параметр в *next* равен 98 или указывает на более низкое значение, то возвращается следующее по порядку целое число. Если параметр равен 99, это значит, что мы наткнулись на конец буфера, поэтому возвращается 0.

Должен быть способ «усыплять» любой из процессов, если он не может продолжаться. Для этого разработчики Java включили в класс *Thread* специальные процедуры *suspend* (отключение) и *resume* (возобновление). Они используются в листинге 6.1.

А теперь рассмотрим саму программу для производителя и потребителя. Сначала производитель порождает новое простое число (шаг P1). Обратите внимание на строку `m.MAX_PRIME`. Префикс `m.` здесь указывает, что имеется в виду `MAX_PRIME`, определенный в классе *m*. По той же причине этот префикс нужен для *in*, *out*, *buffer* и *next*.

Затем производитель проверяет (шаг P2), не находится ли *in* ниже *out*. Если да (например, *in*=62 и *out*=63), то буфер заполнен и производитель вызывает процедуру *suspend* в P2. Если буфер не заполнен, туда помещается новое простое число (шаг P3) и значение *in* увеличивается (шаг P4). Если новое значение *in* на 1 больше значения *out* (шаг P5) (например, *in*=17, *out*=16), значит, *in* и *out* были равны перед тем, как увеличилось значение *in*. Производитель делает вывод, что буфер был пуст и что потребитель не функционировал (находился в режиме ожидания) и в данный момент тоже не функционирует. Поэтому производитель вызывает процедуру *resume*, чтобы возобновить работу потребителя (шаг P5). Наконец, производитель начинает искать следующее простое число.

Листинг 6.1. Параллельная обработка с состоянием гонок

```
public class m {
    final public static int BUF_SIZE = 100;                    // буфер от 0 до 99
    final public static long MAX_PRIME=100000000000L;          //остановиться здесь
    public static int in = 0, out = 0;                         // указатели на данные
    public static long buffer[ ] = new long[BUF_SIZE];         //простые числа хранятся здесь
    public static producer p;                                  //имя производителя
    public static consumer c;                                  //имя потребителя

    public static void main(String args[ ]){                   // основной класс
          p = new producer );                                  //создание производителя
          c = new consumer );                                  //создание потребителя
          p.start();                                           //запуск производителя
          c.start();                                           //запуск потребителя
       }
       //Это утилита для циклического увеличения in и out
       public static int next(int k) {if (k < BUF_SIZE - 1) return(k+1); else return(0);}
}
class producer extends Thread {                                //класс производителя
       public void run() {                                     //код производителя
          long prime = 2;                                      // временная переменная

          while (prime < m.MAX__PRIME) {
             prime = next_prime(prime);                              //шаг P1
             if (m.next(m.in) == m.out) suspend();                   //шаг P2
             m. buffer[m. in] = prime;                               //шаг P3
             m.in = m.next(m.in);                                    //шаг P4
             if (m.next(m.out) == m.in) m.c.resume();                //шаг P5
          }
       }

       private long next_prime(long prime){ ...}      //функция, вычисляющая следующее число

}
class consumer extends Thread {                                //класс потребителя
       public void run() {                                     // код потребителя
          long emirp = 2;                                      // временная переменная
          while (emirp < m.MAX_PRIME) {
             if (m.in == m.out) suspend();                           //шаг C1
             emirp = m.buffer[m.out];                                //шаг C2
             m.out = m.next(m.out);                                  //шаг C3
             if (m.out == m.next(m.next(m.in))) m.p.resume();        //шаг C4
             System.out.println(emirp);                              //шаг C5
          }
       }
}
```

Программа потребителя по структуре очень проста. Сначала производится проверка (шаг C1), чтобы узнать, пуст ли буфер. Если он пуст, то потребителю ничего не нужно делать, поэтому он отключается. Если буфер не пуст, то потребитель удаляет из него следующее число для печати (шаг C2) и увеличивает значение *out*. Если после этого *out* стало на две позиции выше *in*, значит, до этого *out* было на одну позицию выше *in*. Так как *in*=*out*−1 — это условие заполненности буфера, значит, производитель не работает и потребитель должен вызвать процедуру *resume*. После этого число выводится на печать, и весь цикл повторяется снова.

К сожалению, такая программа содержит ошибку (рис. 6.23). Напомним, что эти два процесса работают асинхронно и с разными скоростями, которые, к тому же, могут меняться. Рассмотрим случай, когда в буфере осталось только одно число в элементе 21, и *in*=22, а *out*=21 (см. рис. 6.23, *а*). Производитель на шаге P1 ищет простое число, а потребитель на шаге C5 печатает число из позиции 20. Потребитель заканчивает печатать число, совершает проверку на шаге C1 и забирает последнее число из буфера на шаге C2. Затем он увеличивает значение *out*. В этот момент и *in* и *out* равны 22. Потребитель печатает число, а затем переходит к шагу C1, на котором он вызывает *in* и *out* из памяти, чтобы сравнить их (рис. 6.23, *б*).

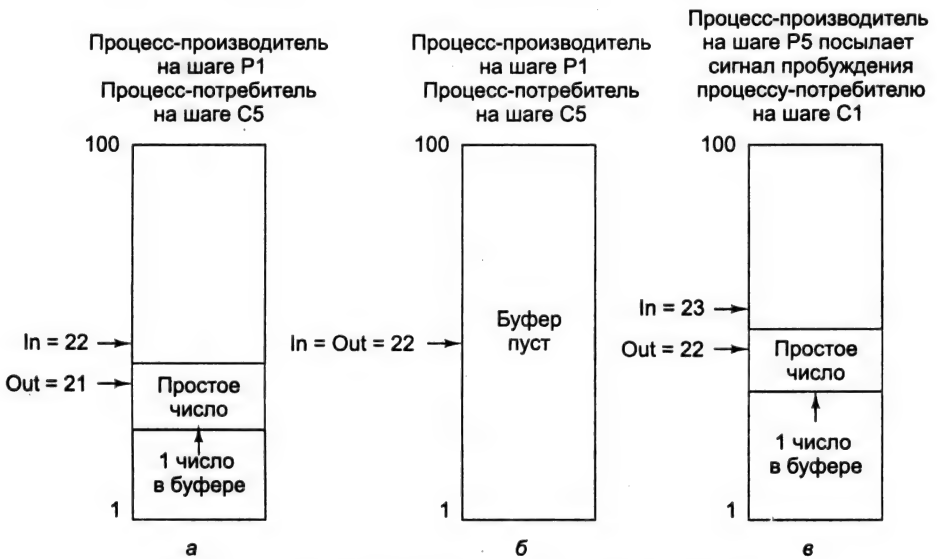


Рис. 6.23. Ситуация, при которой механизм взаимодействия производителя и потребителя не работает

В этот момент, после того как потребитель вызвал *in* и *out*, но еще до того как он сравнил их, производитель находит следующее простое число. Он помещает это простое число в буфер на шаге P3 и увеличивает *in* на шаге P4. Теперь *in*=23, а *out*=22. На шаге P5 производитель обнаруживает, что *in*=*next*(*out*). Иными словами, *in* на единицу больше *out*, а это значит, что в буфере в данный момент находится один элемент.

Исходя из этого, производитель делает неверный вывод, что потребитель отключен, и вызывает процедуру *resume* (рис. 6.23, *в*). На самом деле потребитель все это время продолжал работать, поэтому вызов процедуры *resume* оказался ложным. Затем производитель начинает искать следующее простое число.

В этот момент потребитель продолжает работать. Он уже вызвал *in* и *out* из памяти, перед тем как производитель поместил последнее число в буфер. Так как *in*=22 и *out*=22, потребитель отключается. К этому моменту производитель находит следующее простое число. Он проверяет указатели и обнаруживает, что *in*=24, а *out*=22. Из этого он делает заключение, что в буфере находится 2 числа (что соответствует действительности) и что потребитель функционирует (что неверно).

Производитель продолжает цикл. В конце концов он заполняет буфер и отключается. Теперь оба процесса отключены и будут находиться в таком состоянии до скончания веков.

Сложность здесь в том, что между моментом, когда потребитель вызывает *in* и *out*, и моментом, когда он отключается, производитель, обнаружив, что *in*=*out*+1, и предположив, что потребитель отключен (хотя на самом деле он еще продолжает функционировать), вызывает процедуру *resume*, чего не нужно делать, поскольку потребитель функционирует. Такая ситуация называется **состоянием гонок**, поскольку успех процедуры зависит от того, кто выиграет гонку по проверке *in* и *out*, после того как значение *out* увеличилось.

Проблема состояния гонок хорошо известна. Она была настолько серьезна, что через несколько лет после появления Java компания Sun изменила класс *Thread* и убрала вызовы процедур *suspend* и *resume*, поскольку они очень часто приводили к состоянию гонок. Предложенное решение было основано на изменении языка, но поскольку мы изучаем операционные системы, мы обсудим другое решение, которое используется во многих операционных системах, в том числе UNIX и NT.

Синхронизация процесса с использованием семафоров

Проблему состояния гонок можно разрешить по крайней мере двумя способами. Первый способ — снабдить каждый процесс специальным битом ожидания пробуждения. Если процесс, который функционирует в данный момент, получает сигнал «пробуждения», то этот бит устанавливается. Если процесс отключается в тот момент, когда этот бит установлен, он немедленно перезапускается, а бит сбрасывается. Данный бит сохраняет сигнал пробуждения для будущего использования.

Этот метод решает проблему состояния гонок только в том случае, если у нас есть всего 2 процесса. В общем случае при наличии n процессов он не работает. Конечно, каждому процессу можно приписать n–1 таких битов ожидания пробуждения, но это неудобно.

Дейкстра [31] предложил другое решение этой проблемы. Где-то в памяти находятся две переменные, которые могут содержать неотрицательные целые числа. Эти переменные называются **семафорами**. Операционная система предоставляет два системных вызова, *up* и *down*, которые оперируют семафорами. *Up* прибавляет 1 к семафору, а *down* отнимает 1 от семафора. Если операция *down* совершается над семафором, значение которого больше 0, этот семафор уменьшается на 1 и процесс продолжается. Если значение семафора равно 0, то операция *down* не может завершиться. Тогда данный процесс отключается до тех пор, пока другой процесс не выполнит операцию *up* над этим семафором.

Команда *up* проверяет, не равен ли семафор нулю. Если он равен 0 и другой процесс находится в режиме ожидания, то семафор увеличивается на 1. После этого процесс, который «спит», может завершить операцию *down*, установив семафор на 0. Теперь оба процесса могут продолжать работу. Если семафор не равен 0, команда *up* просто увеличивает его на 1. Семафор позволяет сохранять сигналы пробуждения, так что они не пропадут зря. У семафорных команд есть одно важное свойство: если один из процессов начал выполнять команду над семафором, то

другой процесс не может получить доступ к этому семафору до тех пор, пока первый не завершит выполнение команды или не будет приостановлен при попытке выполнить команду *down* над 0. В таблице 6.5 изложены важные свойства системных вызовов *up* и *down*.

Таблица 6.5. Результаты операций над семафором

Команда	Семафор = 0	Семафор > 0
Up	Семафор = семафор + 1; если другой процесс пытается совершить команду down над этим семафором, теперь он сможет это сделать и продолжить свою работу	Семафор = семафор +1
Down	Процесс останавливается до тех пор, пока другой процесс не выполнит операцию up над этим семафором	Семафор = семафор –1

Как мы уже сказали, в языке Java предусмотрено свое решение проблемы состояния гонок, но мы сейчас обсуждаем операционные системы. Следовательно, нам нужно каким-либо образом выразить использование семафоров в языке Java. Мы предположим, что были написаны две процедуры, *up* и *down*, которые совершают системные вызовы *up* и *down* соответственно. Используя в качестве параметров обычные целые числа, мы сможем выразить применение семафоров в программах на языке Java.

В листинге 6.2 показано, как можно устранить состояние гонок с помощью семафоров. В класс *m* добавляются два семафора: *available*, который изначально равен 100 (это размер буфера), и *filled*, который изначально равен 0. Производитель начинает работу с шага P1, а потребитель — с шага C1. Выполнение процедуры *down* над семафором *filled* сразу же приостанавливает работу потребителя. Когда производитель находит первое простое число, он вызывает процедуру *down* с *available* в качестве параметра, устанавливая *available* на 99. На шаге P5 он вызывает процедуру *up* с параметром *filled*, устанавливая *filled* на 1. Это действие освобождает потребителя, который теперь может завершить вызов процедуры *down*. После этого *filled* принимает значение 0, и оба процесса продолжают работу.

А теперь давайте еще раз рассмотрим состояние гонок. В определенный момент *in*=22, а *out*=21, производитель находится на шаге P1, а потребитель — на шаге C5. Потребитель завершает действие и переходит к шагу C1, который вызывает процедуру *down*, чтобы выполнить ее над семафором *filled*, который до вызова имел значение 1, а после вызова принял значение 0. Затем потребитель берет последнее число из буфера и выполняет процедуру *up* над *available*, после чего *available* принимает значение 100. Потребитель печатает это число и переходит к шагу C1.

Как раз перед тем, как потребитель может вызвать процедуру *down*, производитель находит следующее простое число и быстро выполняет шаги P2, P3 и P4.

В этот момент *filled*=0. Производитель собирается выполнить над ним команду up, а потребитель собирается вызвать процедуру *up*. Если потребитель выполнит команду первым, то он будет приостановлен до тех пор, пока производитель не освободит его (вызвав процедуру *up*). Если же первым будет производитель, то семафор примет значение 1 и потребитель вообще не будет приостановлен. В обоих случаях сигнал пробуждения не пропадет. Именно для этого мы и ввели в программу семафоры.

Операции над семафорами неделимы. Если операция над семафором уже началась, то никакой другой процесс не может использовать этот семафор до тех пор, пока первый процесс не завершит операцию или пока он не будет приостановлен. Более того, при наличии семафоров сигналы пробуждения не пропадают. А вот операторы `if` в листинге 6.1 делимы. Между проверкой условия и выполнением нужного действия другой процесс может послать сигнал пробуждения.

В сущности, проблема синхронизации была устранена путем введения неделимых системных прерываний *up* и *down*. Чтобы эти операции были неделимыми, операционная система должна запретить двум и более процессам использовать один семафор одновременно. Если был сделан системный вызов *up* или *down*, ни один другой код пользователя не будет запущен, пока данный вызов не завершится. Для этого обычно во время выполнения операций над семафорами вводится запрет на прерывания.

Технология с использованием семафоров работает для произвольного количества процессов. Несколько процессов могут «спать», не завершив системный вызов *down* на одном и том же семафоре. Когда какой-нибудь другой процесс выполнит процедуру *up* на том же семафоре, один из ждущих процессов может завершить вызов *down* и продолжить работу. Семафор сохраняет значение 0, и другие процессы продолжают ждать.

Поясним это на другом примере. Представьте себе 20 баскетбольных команд. Они играют 10 партий (процессов). Каждая игра происходит на отдельном поле. Имеется большая корзина (семафор) для баскетбольных мячей. К сожалению, есть только 7 мячей. В каждый момент в корзине находится от 0 до 7 мячей (семафор принимает значение от 0 до 7). Помещение мяча в корзину — это операция *up*, поскольку она увеличивает значение семафора. Извлечение мяча из корзины — это операция *down*, поскольку она уменьшает значение семафора.

В самом начале один игрок от каждого поля посылается к корзине за мячом. Семерым из них удается получить мяч (завершить операцию *down*); оставшиеся трое вынуждены ждать мяч. Их игры временно приостановлены. В конце концов одна из партий заканчивается и мяч возвращается в корзину (выполняется операция *up*). Эта операция позволяет одному из трех оставшихся игроков получить мяч (закончить незавершенную операцию *down*) и продолжить игру. Оставшиеся две партии остаются приостановленными до тех пор, пока еще два мяча не возвратятся в корзину. Когда эти два мяча положат в корзину (то есть будет выполнено еще две операции *up*), можно будут продолжить последние две партии.

Листинг 6.2. Параллельная обработка с использованием семафоров

```
public class m {
    final public static int BUF_SIZE = 100;                  //буфер от 0 до 99
    final public static long MAX_PRIME=100000000000L;        //остановиться здесь
    public static int in = 0, out = 0;                       //указатели на данные
    public static long buffer[ ] = new long[BUF_SIZE];       //здесь хранятся простые числа

    public static producer p;                                //имя производителя
    public static consumer c;                                //имя потребителя
    public static int filled = 0, available = 100;           //семафоры
```

```
    public static void main(String args[ ]) {          //основной клласс
        p = new producer();                            //создание производителя
        c = new consumer();                            //создание потребителя
        p.start();                                     //запуск производителя
        c.start();                                     //запуск потребителя
    }
    // Это утилита для циклического увеличения in и out
    public static int next(int k) {if (k < BUF_SIZE - 1) return(k+1); else return(0);}
}
class producer extends Thread {                        //класс производителя
    native void up(int s); native void down(int s);    //процедуры над семафорами
        public void run() {                            //код производителя
            long prime = 2;                            //временная переменная

            while (prime < m.MAX_PRIME) {
                prime = next_prime(prime);             //шаг P1
                down(m.available);                     //шаг P2
                m.buffer[m.in] = prime;                //шаг P3
                m.in = m.next(m.in);                   //шаг P4
                up(m.filled);                          //шаг P5
        }
    }

private long next_prime(long prime){ ...}              //функция, которая выдает
                                                         следующее число
}

class consumer extends Thread {                        //класс потребителя
        native void up(int s); native void down(int s); //процедуры над семафорами
        public void run() {                            //код потребителя
            long emirp = 2;                            //временная переменная

            while (emirp < m.MAX_PRIME) {
                down(m.filled);                        //шаг C1
                emirp = m.buffer[m.out];               //шаг C2
                m.out = m.next(m.out);                 //шаг C3
                up(m. available);                      //шаг C4
                System.out.println(emirp);             //шаг C5
            }
        }
}
```

Примеры операционных систем

В этом разделе мы продолжим обсуждать Pentium II и Ultra SPARC II. Мы рассмотрим операционные системы, которые используются на этих процессорах.

Для Pentium II мы возьмем Windows NT (для краткости мы будем называть эту операционную систему NT); для UltraSPARC II — UNIX. Мы начнем наш разговор с UNIX, поскольку эта система гораздо проще NT. Кроме того, операционная система UNIX была разработана раньше и сильно повлияла на NT, поэтому такой порядок изложения будет более осмысленным.

Введение

В этом разделе мы дадим краткий обзор двух операционных систем (UNIX и NT). При этом мы обратим особое внимание на историю, структуру и системные вызовы.

UNIX

Операционная система UNIX была разработана в компании Bell Labs в начале 70-х годов. Первая версия была написана Кеном Томпсоном (Ken Thompson) на ассемблере для мини-компьютера PDP-7. Затем была написана вторая версия для компьютера PDP-11, уже на языке C. Ее автором был Деннис Ритчи (Dennis Ritchie). В 1974 году Ритчи и Томпсон опубликовали очень важную работу о системе UNIX [120]. За эту работу они были награждены престижной премией Тьюринга Ассоциации вычислительной техники (Ritchie, 1984; Thompson, 1984). После публикации этой работы многие университеты попросили у Bell Labs копию UNIX. Поскольку материнская компания Bell Labs, AT&T была в тот момент регулируемой монополией и ей не разрешалось принимать участие в компьютерном бизнесе, университеты смогли приобрести операционную систему UNIX за небольшую плату.

PDP-11 использовались практически во всех компьютерных научных отделах университетов, и операционные системы, которые пришли туда вместе с PDP-11, не нравились ни профессорам, ни студентам. UNIX быстро заполнил эту нишу. Эта операционная система была снабжена исходными текстами, поэтому люди могли до бесконечности исправлять ее.

Одним из первых университетов, которые приобрели систему UNIX, был Калифорнийский университет в Беркли. Поскольку имелась в наличии полная исходная программа, в Беркли сумели существенно преобразовать эту систему. Среди изменений было портирование этой системы на мини-компьютер VAX, создание виртуальной памяти со страничной организацией, расширение имен файлов с 14 символов до 255, а также включение сетевого протокола TCP/IP, который сейчас используется в Интернете (во многом благодря тому факту, что он был в системе Berkeley UNIX).

Пока в Беркли производились все эти изменения, компания AT&T самостоятельно продолжала разработку UNIX, в результате чего в 1982 году появилась System III, а в 1984 — System V. В конце 80-х годов широко использовались две разные и совершенно несовместимые версии UNIX: Berkeley UNIX и System V. Такое положение, да еще и отсутствие стандартов на форматы программ в двоичном коде сильно препятствовало коммерческому успеху системы UNIX. Поставщики программного обеспечения не могли писать программы для UNIX, ведь не было никакой гарантии, что эти программы будут работать на любой версии UNIX (как было сделано с MS DOS). После долгих споров комиссия стандартов в Институте инженеров по электротехнике и электронике выпустила стандарт **POSIX** (Portable Operating System-IX — интерфейс переносной операционной системы[1]). POSIX также известен как стандарт P1003. Позднее он стал международным стандартом.

[1] POSIX — Portable Operating System Interface for Computer Environments — платформенно-независимый системный интерфейс. — *Примеч. научн. ред.*

Стандарт POSIX разделен на несколько частей, каждая из которых покрывает отдельную область системы UNIX. Первая часть P1003.1 определяет системные вызовы; вторая часть P1003.2 определяет основные обслуживающие программы и т. д. Стандарт P1003.1 определяет около 60 системных вызовов, которые должны поддерживаться всеми соответствующими системами. Это вызовы для чтения и записи файлов, создания новых процессов и т. д. Сейчас практически все системы UNIX поддерживают системные вызовы P1003.1. Однако многие системы UNIX поддерживают и дополнительные системные вызовы, в частности те, которые определены в System V или в Berkeley UNIX. Обычно к набору POSIX добавляется до 100 системных вызовов. Операционная система для машины UltraSPARC II основана на System V. Она называется **Solaris**. Она поддерживает и многие вызовы из системы Berkeley.

В табл. 6.6 приведены категории системных вызовов. Системные вызовы управления файлами и директориями — это самые большие и самые важные категории. Большинство из них относятся к стандарту P1003.1. Остальные происходят из системы System V.

Таблица 6.6. Системные вызовы UNIX

Категория	Примеры
Управление файлами	Открытие, чтение, запись, закрытие и блокировка файлов
Управление директориями	Создание и удаление директорий; перемещение файлов по директориям
Управление процессами	Порождение, завершение, отслеживание процессов и передача сигналов
Управление памятью	Разделение общей памяти между процессами; защита страниц
Вызов/установка параметров	Идентификация пользователя, группы, процесса; установка приоритетов
Даты и периоды времени	Указание на время доступа к файлам; использование датчика временных интервалов; рабочий профиль программы
Работа в сети	Установка/принятие соединения; отправка/получение сообщения
Прочее	Учет использования ресурсов; ограничение на доступный объем памяти; перезагрузка системы

Сфера использования сетей в большей степени относится к Berkeley UNIX, а не к System V. В Беркли было введено понятие **сокет** (конечный пункт сетевой связи). Четырехпроводные стенные розетки, к которым можно подсоединять телефоны, послужили в качестве модели этого понятия. Процесс в системе UNIX может создать сокет, присоединиться к нему и установить связь с сокетом на удаленном компьютере. По этой связи можно пересылать данные в обоих направлениях, обычно с использованием протокола TCP/IP. Поскольку технология сетевой связи десятилетиями применялась в системе UNIX, значительное число серверов в Интернете используют именно UNIX.

Существует много разных вариантов системы UNIX, и каждая из них чем-то отличается от всех остальных, поэтому структуру данной операционной системы описать трудно. Но схема, изображенная на рис. 6.24, применима к большинству

из них. Внизу находятся драйверы устройств, которые защищают систему файлов от аппаратного обеспечения. Изначально каждый драйвер устройства был написан отдельно от всех других и представлял собой независимую единицу. Это привело к многочисленным дублированиям, поскольку многие драйверы должны иметь дело с управлением потоками, исправлением ошибок, приоритетами, отделением данных от команд и т. д. По этой причине Деннис Ритчи изобрел структуру под названием **поток** для написания драйверов в модулях. При наличии потока можно установить двустороннюю связь между пользовательским процессом и устройством аппаратного обеспечения и вставить между ними один или несколько модулей. Пользовательский процесс передает данные в поток, а затем эти данные обрабатываются или передаются дальше каждым модулем до тех пор, пока они не дойдут до аппаратного обеспечения. При передаче данных от аппаратного обеспечения происходит обратный процесс.



Рис. 6.24. Структура типичной системы UNIX

Над драйверами устройств находится система управления файлами. Она управляет именами файлов, директориями, расположением блоков на диске, защитой и выполняет многие другие функции. В системе файлов имеется так называемый **кэш блоков** для хранения недавно считанных с диска блоков, на случай если они понадобятся еще раз. Некоторые системы файлов использовались на протяжении многих лет. Среди них можно назвать быструю файловую систему Berkeley [95] и журналирующие файловые системы [121].

Еще одна часть ядра системы UNIX — структура управления процессами. Она выполняет различные функции, в том числе управляет межпроцессорной связью, которая позволяет разным процессам взаимодействовать друг с другом и синхронизирует работу процессов, чтобы избежать состояния гонок. Для этого существует ряд механизмов. Код управления процессами также управляет планированием работы процессов, которое основано на приоритетах. Кроме того, здесь обрабатываются сигналы прерываний. Наконец, здесь же происходит управление памятью. Большинство систем UNIX поддерживают виртуальную память с подкачкой страниц по требованию, иногда с некоторыми дополнительными особенностями (например, несколько процессов могут разделять общие области адресного пространства).

UNIX изначально должен был быть маленькой системой, чтобы достигнуть увеличения надежности и высокой производительности. Первые версии UNIX были полностью текстовыми и использовали терминалы, которые могли отображать 24 или 25 строк по 80 символов ASCII-кодов. Пользовательским интерфейсом управляла программа, так называемая **оболочка**, которая предоставляла интерфейс командной строки. Поскольку оболочка не являлась частью ядра, было легко добавлять новые оболочки в UNIX, и с течением времени было придумано несколько черезвычайно сложных оболочек.

Позднее, когда появились графические терминалы, в Массачусетском технологическом институте для UNIX была разработана система **X Windows**. Еще позже полностью доработанный **графический интерфейс пользователя** под названием **Motif** был установлен поверх X Windows. Поскольку требовалось сохранить маленькое ядро, практически весь код системы X Windows и Motif работают в пользовательском режиме вне ядра.

Windows NT

Первая машина IBM PC, выпущенная в 1981 году, была оснащена 16-битной операционной системой индивидуального пользования, работающей в реальном режиме, с командной строкой. Она называлась MS-DOS 1.0. Эта операционная система состояла из находящейся в памяти программы на 8 Кбайт. Через два года появилась более мощная система на 24 Кбайт — MS-DOS 2.0. Она содержала процессор командной строки (оболочку) с рядом особенностей, заимствованных из системы UNIX. В 1984 году компания IBM выпустила машину PC/AT с операционной системой MS-DOS 3.0, размер которой к тому моменту составлял 36 Кбайт. С годами у системы MS-DOS появлялись все новые и новые особенности, но она при этом оставалась системой с командной строкой.

Вдохновленная успехом Apple Macintosh, компания Microsoft решила создать графический пользовательский интерфейс, который она назвала **Windows**. Первые три версии Windows, включая систему Windows 3.x, были не настоящими операционными системами, а графическими пользовательскими интерфейсами на базе MS-DOS. Все программы работали в одном и том же адресном пространстве, и ошибка в любой из них могла привести к остановке всей системы.

В 1995 году появилась система Windows 95, но это не устранило MS-DOS, хотя MS-DOS уже представляла собой новую версию 7.0. Windows 95 и MS-DOS 7.0 в совокупности включали в себя особенности развитой операционной системы, в том числе виртуальную память, управление процессами и мультипрограммирование. Однако операционная система Windows 95 не была полностью 32-битной программой. Она содержала большие куски старого 16-битного кода и все еще использовала файловую систему MS-DOS со всеми ограничениями. Единственным изменением в системе файлов было добавление длинных имен файлов (ранее в MS-DOS длина имен файлов была не более 8+3 символов).

Даже при выпуске Windows 98 в 1998 году система MS-DOS все еще присутствовала (на этот раз версия 7.1) и включала 16-битный код. Система Windows 98 не очень отличалась от Windows 95, хотя часть функций перешла от MS-DOS к Windows и формат дисков, подходящий для дисков большего размера, стал стандартным. Основным различием был пользовательский интерфейс, который объединил рабочий стол, Интернет, телевидение и сделал систему более закрытой. Именно

это и привлекло внимание судебного департамента США, который тогда подал на компанию Microsoft в суд, обвинив ее в незаконном монополизме.

Во время всех этих преобразований компания Microsoft разрабатывала совершенно новую 32-битную операционную систему, которая была написана заново с нуля. Эта новая система называлась **Windows New Technology** (новая технология) или **Windows NT**[1]. Изначально предполагалось, что она заменит все другие операционные системы компьютеров на базе процессоров Intel, но она очень медленно распространялась и позднее была переориентирована на более дорогостоящие компьютеры. Постепенно она стала пользоваться популярностью и в других кругах.

NT продается в двух вариантах: для серверов и для рабочих станций. Эти две версии практически идентичны и выработаны из одного исходного кода. Первая версия предназначена для локальных файловых серверов и серверов для печати и имеет более сложные особенности управления, чем версия для рабочих станций, которая предназначена для настольных вычислений одного пользователя. Существует особый вариант версии для серверов, предназначенный для больших сайтов. Различные версии настраиваются по-разному, и каждая из них оптимизирована для ожидаемого окружения. Во всем остальном эти версии сходны. Практически все выполняемые файлы идентичны для всех версий. Система NT сама определяет свою версию по специальной переменной во внутренней структуре данных (системный реестр). Пользователям запрещено изменять эту переменную и таким образом превращать дешевую версию для рабочей станции в более дорогостоящую версию для сервера или в версию для предприятия. В дальнейшем мы не будем заострять внимание на различиях.

MS-DOS и все предыдущие версии Windows были рассчитаны на одного пользователя. NT поддерживает мультипрограммирование, поэтому на одной и той же машине в одно и то же время могут работать несколько пользователей[2]. Например, сетевой сервер позволяет нескольким пользователям входить в систему по сети одновременно, причем каждый из них получает доступ к своим собственным файлам.

NT представляет собой реальную 32-битную операционную систему с мультипрограммированием. Она поддерживает несколько пользовательских процессов, каждый из которых имеет в своем распоряжении полное 32-битное виртуальное адресное пространство с подкачкой страниц по требованию. Кроме того, сама система написана как 32-битный код.

NT, в отличие от Windows 95, имеет модульную структуру. Она состоит из небольшого ядра, которое работает в привилегированном режиме, и нескольких обслуживающих процессов, работающих в пользовательском режиме. Пользо-

---

[1] Работы над операционной системой, получившей впоследствии название Windows NT, начались в рамках совместного проекта по созданию новой 32-битной версии операционной системы OS/2, который одно время осуществляли компании IBM и Microsoft после ряда неудач с предыдущей 16-битной версией системы OS/2. Этот проект, ориентированный на возможности микропроцессора i386, начался в 1989 году, однако уже в следующем году пути этих компаний разошлись, и Microsoft, продолжившая работу над системой OS/2 v.3.0, затем дала ей имя Windows NT, желая этим показать и использование единого графического интерфейса со своими популярными одноименными оболочками, и дистанцирование от компании IBM. — *Примеч. научн. ред.*

[2] Необходимо заметить, что, в отличие от UNIX, Windows NT не позволяет нескольким пользователям одновременно работать с компьютером, поскольку это однотерминальная система, тогда как UNIX — это мультитерминальная операционная система. Однако по сети с ней могут одновременно взаимодействовать несколько пользователей, работающих на своих компьютерах. — *Примеч. научн. ред.*

вательские процессы взаимодействуют с обслуживающими процессами с применением модели клиент—сервер: клиент посылает запрос серверу, а сервер выполняет работу и отправляет результат клиенту. Модульная структура позволяет переносить систему NT на некоторые компьютеры не из семейства Intel (DEC Alpha, IBM Power PC и SGI MIPS). Однако из соображений повышения производительности начиная с NT 4.0 большая часть системы была перенесена обратно в ядро.

Можно до бесконечности долго рассказывать, какова структура NT и каков ее интерфейс. Поскольку нас в первую очередь интересует виртуальная машина, представленная различными операционными системами, мы кратко расскажем о структуре системы, а затем перейдем к интерфейсу.

Структура NT показана на рис. 6.25. Она состоит из ряда модулей, которые расположены по уровням. Их совместная работа реализует операционную систему. Каждый модуль выполняет определенную функцию и имеет определенный интерфейс с другими модулями. Практически все модули написаны на языке C, хотя часть графического интерфейса написана на C++, а кое-что из самых нижних уровней — на ассемблере.

В самом низу расположен **уровень аппаратных абстракций**. Он должен снабжать операционную систему абстрактными аппаратными устройствами, лишенными всех недостатков, которых у реального аппаратного обеспечения в избытке. К моделируемым устройствам относятся кэш-память вне кристалла, тактовые генераторы, шины ввода-вывода, контроллеры прерываний и контроллеры прямого доступа к памяти. Если эти устройства представить перед операционной системой в идеализированном виде, то это упростит перенос NT на другие аппаратные платформы, поскольку большая часть необходимых изменений концентрируется в одном месте.



Рис. 6.25. Структура Windows NT

Над уровнем аппаратных абстракций расположен уровень, содержащий микроядро и драйверы устройств. Микроядро и все драйверы устройств имеют прямой доступ к аппаратному обеспечению, поскольку они содержат зависимый от аппаратного обеспечения код.

Микроядро поддерживает примитивные объекты ядра, обработку прерываний, ловушек, исключений, синхронизацию процессов, синхронизацию работы процессоров в многопроцессорных системах и управление временем. Основная задача этого уровня — сделать остальную часть операционной системы полностью независимой от аппаратного обеспечения и, следовательно, высокомобильной. Микроядро постоянно находится в основной памяти и никуда не передается, хотя оно временно может передать управление прерываниям ввода-вывода.

Каждый **драйвер устройств** может управлять одним или несколькими устройствами ввода-вывода. Кроме того, драйвер устройств может выполнять какие-то функции, не связанные с конкретным устройством, например шифровку потока данных или даже обеспечение доступа к структурам данных ядра. Так как пользователи имеют возможность устанавливать новые драйверы устройств, они могут повлиять на ядро и испортить всю систему. По этой причине драйверы нужно писать с особой осторожностью.

Над микроядром и драйверами устройств находится исполняющая система. **Исполняющая система** — независимая архитектура, поэтому ее можно переносить на другие машины. Она состоит из трех уровней.

Самый нижний уровень содержит файловые системы и диспетчер объектов. **Файловые системы** управляют использованием файлов и директорий. **Диспетчер объектов** управляет объектами, известными ядру (процессами, потоками, директориями, семафорами, устройствами ввода-вывода, тактовыми генераторами и т. п.). Эта программа также управляет пространством имен, куда можно помещать новые объекты, чтобы обращаться к ним позже в случае необходимости.

Следующий уровень состоит из 6 основных частей, как показано на рис. 6.25. **Диспетчер ввода-вывода** обеспечивает структуру для управления устройствами ввода-вывода, а также общими службами ввода-вывода. Диспетчер ввода-вывода использует службы файловой системы, которая, в свою очередь, использует драйверы устройств, а также службы диспетчера объектов.

Диспетчер кэш-памяти хранит в памяти блоки с диска, которые недавно использовались, чтобы повысить скорость доступа к ним, если они понадобятся снова. Диспетчер кэш-памяти должен вычислять, какие блоки могут понадобиться снова, а какие — нет. Можно конфигурировать NT с несколькими системами файлов. В этом случае диспетчер кэш-памяти работает на все системы файлов, поэтому отдельный диспетчер для каждой из них не нужен. Если требуется какой-либо блок диска, диспетчеру кэш-памяти посылается сигнал выдать этот блок. Если данного блока нет, диспетчер вызывает соответствующую систему файлов, чтобы получить этот блок. Поскольку файлы могут быть отображены на адресные пространства процессов, диспетчер кэш-памяти должен взаимодействовать с модулем управления виртуальной памятью, чтобы обеспечить необходимую согласованность.

Модуль управления виртуальной памятью реализует архитектуру виртуальной памяти с подкачкой страниц по требованию. Он управляет отображением виртуальных страниц на физические страничные кадры. Он вводит дополнительные

правила защиты, которые ограничивают доступ каждого процесса только к тем страницам, которые принадлежат его адресному пространству. Он также обрабатывает некоторые системные вызовы, которые связаны с виртуальной памятью.

Диспетчер процессов и потоков управляет процессами и потоками, в том числе их созданием и удалением.

Диспетчер безопасности предоставляет механизмы безопасности NT, которые удовлетворяют требованиям Оранжевой книги департамента защиты США. В Оранжевой книге определяется огромное количество правил, которым должна удовлетворять система, начиная с пароля и заканчивая тем, что виртуальные страницы должны обнуляться перед повторным использованием.

Интерфейс графических устройств управляет изображением на мониторе и принтерами. Он обеспечивает системные вызовы, которые позволяют пользовательским программам записывать информацию на монитор или принтеры независимо от устройств. Он также содержит диспетчер окон и драйверы аппаратных устройств. В версиях NT до NT 4.0 он находился в пользовательском пространстве, но производительность при этом оставляла желать лучшего, поэтому компания Microsoft перенесла его в ядро. Модуль Win32 также управляет многими системными вызовами. Изначально он тоже располагался в пользовательском пространстве, но позднее был перемещен в ядро с целью повышения производительности.

Самый верхний уровень исполняющей системы — **системные службы**. Этот уровень обеспечивает интерфейс с исполняющей системой. Он принимает системные вызовы NT и вызывает другие части исполняющей системы для выполнения.

Вне ядра находятся пользовательские программы и **подсистемы окружения**. Необходимость подсистем окружения объясняется тем, что пользовательские программы не способны непосредственно осуществлять системные вызовы. Поэтому каждая такая подсистема экспортирует определенный набор вызовов функций, которые могут использовать пользовательские программы. На рисунке 6.25 показаны 3 подсистемы окружения: Win32 (для программ NT и Windows 95/98), POSIX (для программ UNIX) и OS/2 для программ OS/2[1].

Приложения Windows используют функции **подсистемы Win32** и взаимодействуют с подсистемой Win32, чтобы совершать системные вызовы. **Подсистема Win32** принимает вызовы функций Win32 и использует модуль **системного интерфейса**, чтобы системные вызовы NT могли выполнять их.

Подсистема POSIX обеспечивает поддержку для приложений UNIX. Она поддерживает только стандарт P1003.1. Это закрытая подсистема. Ее приложения не могут использовать приспособления подсистемы Win32, что сильно ограничивает ее возможности. На практике перенос любой программы UNIX на NT с использованием этой подсистемы почти невозможен. Ее включили в NT только потому, что правительство США потребовало, чтобы операционные системы на компьютерах в правительстве соответствовали стандарту P1003.1. Эта подсистема не является самодостаточной, поэтому для своей работы она использует подсистему Win32, но при этом не передает полный интерфейс Win32 своим пользовательским программам.

[1] Бытует заблуждение, что Windows NT может выполнять различные программы, разработанные для OS/2. Однако на самом деле эта подсистема NT позволяет выполнять только те немногие 16-битные программы OS/2, которые работают исключительно в текстовом режиме. 32-битные программы OS/2 система Windows NT выполнять не может. — *Примеч. научн. ред.*

Функции подсистемы OS/2 тоже ограничены. Возможно, в будущих версиях ее уже не будет. Она также использует подсистему Win32. Существует и подсистема MS DOS (она не показана на рисунке).

Перейдем к обсуждению служб, которые предлагает операционная система NT. Ее интерфейс — это основное средство связи программиста с системой. К сожалению, компания Microsoft не опубликовала полный список системных вызовов NT, и кроме того, она меняет их от выпуска к выпуску. При таких обстоятельствах практически невозможно написание программ, которые непосредственно совершают системные вызовы.

Зато компания Microsoft определила набор вызовов **Win32 API (Application Programming Interface — прикладной программный интерфейс)**. Это библиотечные процедуры, которые либо совершают системные вызовы, чтобы выполнить определенные действия, либо в некоторых случаях выполняют некоторые действия прямо в библиотечной процедуре пользовательского пространства или подсистеме Win32. Вызовы Win32 API не меняются при создании новых версий.

Однако, кроме этого, существуют вызовы NT API, которые могут менятся в новых версиях NT. Так как вызовы Win32 API задокументированы и более стабильны, мы сосредоточим наше внимание именно на них, а не на системных вызовах NT.

В системах Win32 API и UNIX применяются совершенно разные подходы. В UNIX все системные вызовы общеизвестны и формируют минимальный интерфейс: удаление хотя бы одного из них изменит функционирование операционной системы. Подсистема Win32 обеспечивает очень полный интерфейс. Здесь часто одно и то же действие можно выполнить тремя или четырьмя разными способами. Кроме того, Win32 включает в себя много функций, которые не являются системными вызовами (например, копирование целого файла).

Многие вызовы Win32 API создают объекты ядра того или иного типа (файлы, процессы, потоки, каналы и т. п.). Каждый вызов, создающий объект ядра, возвращает вызывающей программе результат, который называется **идентификатором (handle)**. Этот идентификатор впоследствии может использоваться для выполнения операций над объектом. Для каждого процесса существует свой идентификатор. Он не может передаваться другому процессу и использоваться там (дескрипторы файла в UNIX тоже нельзя передавать другому процессу). Однако при определенных обстоятельствах можно продублировать идентификатор, передать его другим процессам и разрешить им доступ к объектам, которые принадлежат другим процессам. Каждый объект имеет связанный с ним дескриптор защиты, который сообщает, кому разрешено или запрещено совершать те или иные операции над объектом.

Операционную систему NT иногда называют объектно-ориентированной, поскольку оперировать с объектами ядра можно только с помощью вызова процедур (функций API) по их идентификаторам. С другой стороны, она не обладает такими основными свойствами объектно-ориентированной системы, как наследование и полиморфизм.

Win32 API имеется и в системе Windows 95/98 (а также в операционной систме Windows CE), правда, с некоторыми исключениями. Например, Windows 95/98 не имеет защиты, поэтому те вызовы API, которые связаны с защитой, просто возвращают код ошибки. Кроме того, для имен файлов в NT используется набор

символов Unicode, которого нет в Windows 95/98. Существуют различия в параметрах для некоторых вызовов API. В системе NT, например, все координаты экрана являются 32-битными числами, а в системе Windows 95/98 используются только младшие 16 битов (для совместимости с Windows 3.1). Существование набора вызовов Win32 API на нескольких разных операционных системах упрощает перенос программ между ними, но при этом кое-что удаляется из основной системы вызовов. Различия между Windows 95/98 и NT изложены в табл. 6.7.

Таблица 6.7. Некоторые различия между версиями Windows

Характеристика	Windows 95/98	NT 5.0
Win32 API	Да	Да
Полностью 32-битная система	Нет	Да
Защита	Нет	Да
Отображение защищенных файлов	Нет	Да
Отдельное адресное пространство для каждой программы MS-DOS	Нет	Да
Plug and Play	Да	Да
Unicode	Нет	Да
Процессор	Intel 80×86	80×86, Alpha
Многопроцессорная поддержка	Нет	Да
Реентерабельная программа (допускающая повторное вхождение) внутри операционной системы	Нет	Да
Пользователь может сам написать некоторые важные части операционной системы	Да	Нет

Примеры виртуальной памяти

В этом разделе мы поговорим о виртуальной памяти в системах UNIX и NT. С точки зрения программиста они во многом сходны.

Виртуальная память UNIX

Модель памяти в системе UNIX довольно проста. Каждый процесс имеет три сегмента: код, данные и стек, как показано на рис. 6.26. В машине с линейным адресным пространством код обычно располагается в нижней части памяти, а за ним следуют данные. Стек помещается в верхней части памяти. Размер кода фиксирован, а данные и стек могут увеличиваться или уменьшаться. Такую модель легко реализовать практически на любой машине. Она используется в операционной системе Solaris.

Более того, если машина содержит страничную память, то все адресное пространство может быть разбито на страницы, а пользовательские программы этого не знают. Единственное, что им будет известно, — то, что размер программы может превышать размер физической памяти машины. Системы UNIX, у которых нет страничной организации памяти, обычно перекачивают целые процессы между памятью и диском, чтобы сколь угодно большое число процессов работало в режиме разделения времени.

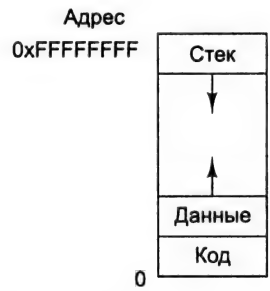


Рис. 6.26. Адресное пространство одного процесса UNIX

Описание, данное выше (виртуальная память с подкачкой страниц по требованию), в целом подходит для Berkeley UNIX. Однако Sysytem V (и Solaris) имеет некоторые особенности, позволяющие пользователям управлять виртуальной памятью. Самой важной является способность процесса отображать файл или часть файла на часть его адресного пространства. Например, если файл в 12 Кбайт отображается на виртуальный адрес 144 K, то в ячейке с адресом 144 K будет находиться первое слово этого файла. Таким образом, можно осуществлять ввод-вывод файла без применения системных вызовов. Поскольку размер некоторых файлов может превышать размер виртуального адресного пространства, можно отображать не весь файл, а только его часть. Чтобы осуществить отображение, сначала нужно открыть файл и получить дескриптор файла *fd* (file descriptor). Дескриптор используется для идентификации файла, который нужно отобразить. Затем процесс совершает вызов

```
paddr=mmap(virtual_address,length,protection,flags,fd,file_offset)
```

который отображает *length*, начиная с *file_offset* в файле, в виртуальное адресное пространство, начиная с *virtual_address*. Параметр *flags* требует, чтобы система выбрала виртуальный адрес, который затем возвращается в *paddr*. Отображаемая область должна содержать целое число страниц и должна быть выровнена в границах страницы. Параметр *protection* определяет разрешение на чтение, запись и выполнение (в любой комбинации). Отображение можно в дальнейшем удалить с помощью команды `unmap`.

В один и тот же файл можно одновременно отображать несколько процессов. Есть два варианта разделения общих страниц. В первом случае разделяются все страницы, поэтому записи, призводимые одним процессом, видны всем другим процессам. Эта возможность обеспечивает между процессами тракт с высокой пропускной способностью. Во втором случае страницы разделяются всеми процессами до тех пор, пока какой-нибудь процесс не изменит их. Как только один из процессов пытается произвести запись в страницу, он получает ошибку защиты, в результате чего операционная система выдает ему копию этой страницы, в которую можно производить запись. Такая схема используется в том случае, когда для каждого из нескольких процессов нужно создать иллюзию, что только он отображается в файл.

Виртуальная память Windows NT

В NT каждый пользовательский процесс имеет свое собственное виртуальное адресное пространство. Длина виртуального адреса составляет 32 бита, поэтому каждый процесс имеет 4 Гбайт виртуального адресного пространства. Нижние

2 Гбайт предназначены для кода и данных процесса; верхние 2 Гбайт разрешают ограниченный доступ к памяти ядра. Исключение составляют версии NT для предприятий, в которых разделение памяти может быть другим: 3 Гбайт — для пользователя и 1 Гбайт — для ядра. Виртуальное адресное пространство с подкачкой страниц по требованию содержит страницы фиксированного размера (4 Кбайт на машине Pentium II).

Каждая виртуальная страница может находиться в одном из трех состояний: она может быть **свободной** (free), **зарезервированной** (reserved) и **выделенной** (committed). Свободная страница в текущий момент не используется, а обращение к ней вызывает ошибку из-за отсутствия страницы. Когда процесс начинается, все его страницы находятся в свободном состоянии до тех пор, пока программа и начальные данные не будут отображены в свое адресное пространство. Если код или данные отображены в страницу, то такая страница является выделенной. Обращение к выделенной странице будет успешным, если страница находится в основной памяти. Если страница отсутствует в основной памяти, то произойдет ошибка и операционной системе придется вызывать нужную страницу с диска. Виртуальная страница может находиться и в зарезервированном состоянии. Это значит, что эта страница недоступна для отображения до тех пор, пока резервирование не будет отменено. Помимо атрибутов состояния страницы имеют и другие атрибуты (например, указывающие на возможность чтения, записи и выполнения). Верхние 64 Кбайт и нижние 64 Кбайт памяти всегда свободны, чтобы можно было отыскивать ошибки указателей (неинициализированные указатели часто равны 0 или –1).

Каждая выделенная страница имеет теневую страницу на диске, где она хранится при отсутствии ее в основной памяти. Свободные и зарезервированные страницы не имеют теневых страниц, поэтому обращения к ним вызывают ошибки из-за отсутствия страницы (система не может вызвать страницу с диска, если этой страницы нет на диске). Теневые страницы на диске сгруппированы в один или несколько страничных файлов. Операционная система следит, в какую часть какого страничного файла отображается каждая виртуальная страница. Файлы с текстами программ имеют теневые страницы; для страниц данных используются специальные страничные файлы.

NT, как и System V, позволяет отображать файлы прямо в области виртуального адресного пространства. Если файл был отображен в адресное пространство, его можно считывать или записывать путем обычных обращений к памяти.

Отображаемые в память файлы реализуются так же, как другие выделенные страницы, только теневые страницы могут находиться в файле на диске, а не в страничном файле. В результате, когда файл отображается, версия в памяти может не совпадать с версией на диске (из-за последних записей в виртуальное адресное пространство). Однако когда отображение файла удаляется, версия на диске обновляется.

NT позволяет отображать два и более процессов в одном файле одновременно, возможно, в разных виртуальных адресах. Путем считывания слов из памяти и записи слов в память процессы могут взаимодействовать друг с другом и передавать данные в обоих направлениях с достаточно высокой скоростью, поскольку никакого копирования не требуется. Разные процессы могут обладать разными разрешениями на доступ. Все процессы, использующие отображенный файл, разделяют

одни и те же страницы, поэтому изменения, произведенные одним из процессов, видны всем остальным процессам, даже если файл на диске еще не был обновлен.

Win32 API содержит ряд функций, которые позволяют процессу открыто управлять виртуальной памятью. Самые важные из этих функций приведены в табл. 6.8. Все они работают в области, состоящей либо из одной страницы, либо из двух или более страниц, последовательно расположенных в виртуальном адресном пространстве.

Таблица 6.8. Основные функции API для управления виртуальной памятью в системе Windows NT

Функция API	Значение
VirtualAlloc	Резервация или выделение области
VirtualFree	Освобождение области или снятие выделения
VirtualProtect	Изменение типа защиты на чтение/запись/выполнение
VirtualQuery	Запрос о состоянии области
VirtualLock	Делает область памяти резидентной (то есть запрещает разбиение на страницы в ней)
VirtualUnlock	Снимает запрет на разбиение на страницы
CreateFileMapping	Создает объект отображения файла и иногда приписывает ему имя
MapViewOfFile	Отображает файл или часть файла в адресное пространство
UnmapViewOfFile	Удаляет отображенный файл из адресного пространства
OpenFileMapping	Открывает ранее созданный объект отображения файла

Первые четыре функции очевидны. Следующие две функции позволяют процессу делать резидентной область памяти размером до 30 страниц и отменять это действие. Это качество может понадобиться программам, работающим в режиме реального времени. Операционная систма устанавливает определенный предел, чтобы процессы не становились слишком поглощающими. В системе NT также имеются функции API, которые позволяют процессу получать доступ к виртуальной памяти другого процесса (они не указаны в табл. 6.8).

Последние 4 функции API предназначены для управления отображаемыми в память файлами. Чтобы отобразить файл, сначала нужно создать объект отображения файла с помощью функции *CreateFileMapping*. Эта функция возвращает идентификатор (handle) объекту отображения файла и иногда еще и вводит в систему файлов имя для него, чтобы другой процесс мог использовать объект. Две функции отображают файлы и удаляют отображение соответственно. Следующая функция нужна для того, чтобы отобразить файл, который в данный момент отображен другим процессом. Таким образом, два и более процессов могут разделять области своих адресных пространств.

Эти функции API являются основными. На них строится вся остальная система управления памятью. Например, существуют функции API для размещения и освобождения структур данных в одной или нескольких «кучах». «Кучи» используются для хранения структур данных, которые создаются и разрушаются. «Кучи» не занимаются «сборкой мусора», поэтому пользовательское программное обеспечение само должно освобождать блоки виртуальной памяти, которые уже не нуж-

ны. («Сборка мусора» — это автоматическое удаление неиспользуемых структур данных.) «Куча» в NT сходна с функцией *malloc* в системах UNIX, но в NT, в отличие от UNIX, может быть несколько независимых «куч».

Примеры виртуального ввода-вывода

Основной задачей любой операционной системы является предоставление служб для пользовательских программ. Главным образом, это службы ввода-вывода для чтения и записи файлов. И UNIX, и NT предлагают широкий спектр служб ввода-вывода. Для большинства системных вызовов UNIX в NT имеется эквивалентный вызов, но обратное не верно, поскольку NT содержит гораздо больше вызовов и каждый из них гораздо сложнее соответствующего вызова в UNIX.

Виртуальный ввод-вывод в системе UNIX

Система UNIX пользовалась большой популярностью во многом благодаря своей простоте, которая, в свою очередь, является прямым результатом организации системы файлов. Обычный файл представляет собой линейную последовательность 8-битных байтов[1] от 0 до максимум $2^{32}-1$ байтов. Сама операционная система не сообщает структуру записей в файлах, хотя многие пользовательские программы рассматривают текстовые файлы в коде ASCII как последовательности строк, каждая из которых завершается переводом строки.

С каждым открытым файлом связан указатель на следующий байт, который нужно считать или записать. Системные вызовы *read* и *write* считывают и записывают данные, начиная с позиции, которую определяет указатель. После операции оба вызова перемещают указатель в другую позицию, передвигая его ровно на столько байтов, сколько было считано или записано. Возможен и случайный доступ к файлам, когда указатель файла устанавливается на определенное значение.

Кроме обычных файлов, система поддерживает специальные файлы, которые используются для доступа к устройствам ввода-вывода. С каждым устройством ввода-вывода обычно связан один или несколько специальных файлов. Считывая информацию из этих файлов и записывая информацию в эти файлы, программа может считывать информацию с устройства ввода-вывода и записывать информацию на устройство ввода-вывода. Так происходит работа с дисками, принтерами, терминалами и многими другими устройствами.

Основные системные вызовы для файлов в UNIX приведены в табл. 6.9. Вызов *creat* (без *e* на конце) используется для создания нового файла. В настоящее время он не является обязательным, поскольку вызов *open* тоже может создавать новый файл. Вызов *unlink* удаляет файл (предполагается, что файл находится только в одной директории).

Вызов *open* используется для открытия существующих файлов, а также для создания новых. Флаг *mode* сообщает, как его открыть (для чтения, для записи и т. д.). Вызов возвращает небольшое целое число, которое называется **дескриптором файла**. Дескриптор файла определяет файл в последующих вызовах. Сам про-

[1] Для многих сейчас слова про 8-битные байты могут показаться странными, однако на самом деле раньше байт мог быть и 5-битным, и 7-битным, и 8-битным. Теперь мы по умолчанию считаем байт состоящим из 8 битов. — *Примеч. научн. ред.*

цесс ввода-вывода осуществляется с помощью процедур *read* и *write*, каждая из которых содержит дескриптор файла (он указывает, какой файл использовать), буфер для данных и число байтов, которое сообщает, какое количество данных нужно передать. Вызов *lseek* используется для перемещения указателя файла, что делает возможным случайный доступ к файлам.

Stat выдает информацию о файле (размер, время последнего доступа, имя владельца и т. п.). *Chmod* изменяет режим защиты файла (например, разрешает или, наоборот, запрещает каким-нибудь пользователям читать его). Наконец, *fcntl* выполняет различные действия над файлами, например блокирование и разблокирование.

Таблица 6.9. Основные системные вызовы UNIX

Системный вызов	Значение
creat(name, mode)	Создает файл; *mode* определяет тип защиты
unlink(name)	Удаляет файл (предполагается, что есть только 1 связь)
open(name, mode)	Открывает или создает файл и возвращает дескриптор файла
close(fd)	Закрывает файл
read (fd, buffer, count)	Считывает байты в количестве *count* в *buffer*
write(fd, buffer, count)	Записывает в файл *count* байтов из *buffer*
lseek(fd, offset, w)	Перемещает указатель файла на *offset* и *w*
stat(name, buffer)	Возвращает информацию о файле
chmod(name, mode)	Изменяет тип защиты файла
fcntl(fd, cmd, j)	Производит различные операции управления (например, блокирует файл или его часть)

В листинге 6.3 показано, как происходит процесс ввода-вывода. Эта программа минимальна и не включает в себя проверку ошибок. Перед тем как войти в цикл, программа открывает существующий файл *data* и создает новый файл *newf*. Каждый вызов возвращает дескриптор файла *infd* и *outfd* соответственно. Следующий параметр в обоих вызовах — биты защиты, которые определяют, что файлы нужно считать и записать соответственно. Оба вызова возвращают дескриптор файла. Если не удалось произвести *open* или *creat*, то возвращается отрицательный дескриптор файла, который сообщает, что вызов не удался.

Листинг 6.3. Фрагмент программы для копирования файла с использованием системных вызовов UNIX. Этот фрагмент написан на языке C, поскольку в языке Java не показываются системные вызовы низкого уровня, а нам нужно их показать

```
/* Открытие дескрипторов файла. */
infd = open("data", 0);
outfd = creat("newf", ProtectionBits);
/* Цикл копирования. */
do{
       count = read(infd, buffer, bytes);
       if (count > 0) write(outfd, buffer, count);
} while (count > 0);

/* Закрытие файлов.*/
close(infd);
close(outfd);
```

Вызов *read* имеет три параметра: дескриптор файла, буфер и число байтов. Данный вызов должен считать нужное число байтов из указанного файла в буфер. Число считанных байтов помещается в *count*. *Count* может быть меньше, чем *bytes*, если файл был слишком коротким. Вызов *write* копирует считанные байты в выходной файл. Цикл продолжается до тех пор, пока входной файл не будет прочитан полностью. Тогда цикл завершается, а оба файла закрываются.

Дескрипторы файлов в системе UNIX представляют собой небольшие целые числа (обычно до 20). Дескрипторы файлов 0, 1 и 2 соответствуют **стандартному вводу**, **стандартному выводу** и **стандартной ошибке** соответственно. Обычно первый из них обращается к клавиатуре, а второй и третий — к дисплею, но пользователь может перенаправить их к файлам. Многие программы UNIX получают входные данные из стандартного устройства ввода и записывают выходные данные в стандартное устройство вывода. Такие программы называются **фильтрами**.

С системой файлов тесно связана система директорий. Каждый пользователь может иметь несколько директорий, а каждая директория может содержать файлы и поддиректории. Система UNIX обычно конфигурируется с главной директорией, так называемым **корневым каталогом**, который содержит поддиректории *bin* (для часто используемых программ), *dev* (для специальных файлов устройств ввода-вывода), *lib* (для библиотек) и *usr* (для пользовательских директорий, как показано на рис. 6.27). В нашем примере директория *usr* содержит поддиректории *ast* и *jim*. Директория *ast* включает в себя два файла (*data* и *foo.c*) и поддиректорию *bin*, в которую входят 4 игры.

Чтобы назвать файл, нужно указать его **путь** из корневого каталога. Путь содержит список всех директорий от корневого каталога к файлу, для разделения директорий используется слэш. Например, путь к файлу *game2* будет таким: /usr/ast/bin/game2. Путь, который начинается с корневого каталога, называется **абсолютным путем**.

В каждый момент времени каждая работающая программа имеет **текущий каталог**. Путь может быть связан с текущим каталогом. В этом случае в начале пути слэш не ставится (чтобы отличить такой путь от абсолютного пути). Такой путь называется **относительным путем**. Если /usr/ast — текущий каталог, то можно получить доступ к файлу game3, используя путь bin/game3. Пользователь может создать **связь** с чужим файлом, используя для этого системный вызов *link*. В нашем примере пути /usr/ast/bin/game3 и /usr/jim/jotto приводят к одному и тому же файлу. Не разрешается применять связи к директориям, чтобы предотвратить циклы в системе директорий. Вызовы *open* и *creat* используют и абсолютные, и относительные пути.

Основные вызовы для оперирования с директориями в системе UNIX приведены в табл. 6.10. *Mkdir* создает новую директорию, а *rmdir* удаляет существующую пустую директорию. Следующие три вызова применяются для чтения элементов директорий. Первый открывает директорию, второй считывает элементы из нее, а третий закрывает директорию. *Chdir* изменяет текущую директорию.

Link создает элемент директории, который указывает на уже существующий файл. Например, элемент /usr/jim/jotto можно создать с помощью вызова

```
link("/usr/ast/bin/game3", "/usr/jim/jotto")
```

или с помощью эквивалентного вызова, используя относительные пути, которые зависят от текущей директории. *Unlink* удаляет элемент директории. Если файл

имеет только одну связь, он удаляется. Если он имеет две и более связей, то он не удаляется. Не имеет никакого значения, была ли удаленная связь изначально созданной, или это копия. Вызов

```
unlink("/usr/ast/bin/game3")
```

делает файл *game3* доступным только через путь /usr/jim/jotto. Вызовы *link* и *unlink* могут использоваться для перемещения файлов из одной директории в другую.

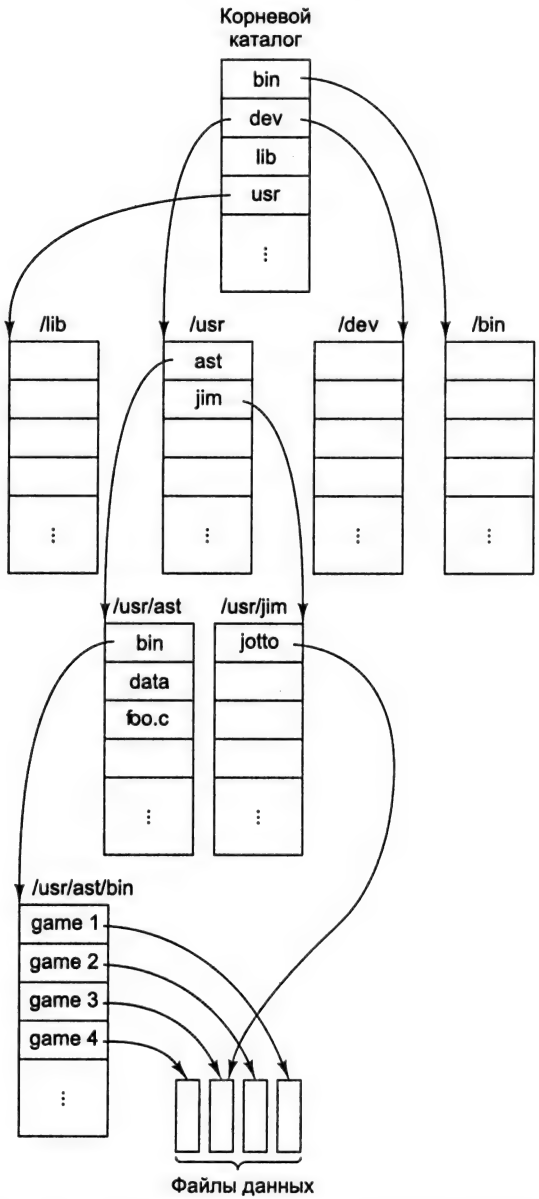


Рис. 6.27. Часть системы директорий в операционной системе UNIX

С каждым файлом (а также с каждой директорией, поскольку директория — это тоже файл) связано битовое отображение, которое сообщает, кому разрешен доступ к этому файлу. Отображение содержит три поля RWX (Read, Write, eXecute — чтение, запись, выполнение). Первое из них контролирует разрешение на чтение, запись и выполнение файлов для их владельца, второе — для других пользователей из группы владельца, а третье — для любых пользователей. Поля RWX R-X --X означают, что владелец файла может читать этот файл, записывать что-либо в него и выполнять его (очевидно, файл является выполняемой программой, иначе не было бы разрешения на его выполнение), другие члены группы могут читать и выполнять его, а посторонние люди — только выполнять. Таким образом, посторонние люди могут использовать эту программу, но не могут ее украсть (скопировать), поскольку им запрещено чтение. Отнесение пользователей к тем или иным группам осуществляется системным администратором, которого обычно называют **привилегированным пользователем**. Привилегированный пользователь имеет право действовать вопреки механизму защиты и считывать, записывать и выполнять любой файл.

Таблица 6.10. Основные вызовы для работы с директориями в системе UNIX

Системный вызов	Значение
mkdir(name, mode)	Создает новую директорию
rmdir(name)	Удаляет пустую директорию
Opendir(name)	Открывает директорию для чтения
readdir(dirpointer)	Читает следующий элемент директории
Closedir(dirpointer)	Закрывает директорию
chdir(dirname)	Изменяет текущий каталог на *dirname*
link(name1, name2)	Создает элемент директории *name2*, указывающий на *name1*
unlink(name)	Удаляет *name* из директории

Теперь рассмотрим, как файлы и директории реализованы в системе UNIX. Более детальное описание см. в [152]. С каждым файлом (и с каждой директорией, поскольку директория — это тоже файл) связан блок информации в 64 байта, который называется **индексным дескриптором (i-node)**. I-node сообщает, кто владеет файлом, что разрешено делать с файлом, где найти данные и т. п. Индексные дескрипторы для файлов расположены или последовательно в начале диска, или, если диск разделен на группы цилиндров, — в начале цилиндра. Индексные дескрипторы снабжены последовательными номерами. Таким образом, система UNIX может обнаружить i-node просто путем вычисления его адреса на диске.

Элемент директории состоит из двух частей: имени файла и номера индексного дескриптора. Когда программа выполняет команду

```
open("foo.c", 0),
```

система ищет текущий каталог для файла «foo.c», чтобы найти номер индексного дескриптора для этого файла. Обнаружив номер индексного дескриптора, программа может считать его и узнать всю информацию об этом файле.

При большей длине пути файла основные шаги, изложенные выше, повторяются несколько раз, пока не будет пройден весь путь. Например, чтобы найти номер индексного дескриптора для пути /usr/ast/data, система сначала ищет корне-

вой каталог для элемента *usr*. Обнаружив индексный дескриптор для *usr*, она может прочитать этот файл (директория в системе UNIX — это тоже файл). В этом файле она ищет элемент *ast* и находит номер индексного дескриптора для файла /usr/ast. Считав информацию о местонахождении директории /usr/ast, система может обнаружить элемент для *data* и, следовательно, номер индексного дескриптора для /usr/ast/data. Найдя номер индексного дескриптора для этого файла, система может узнать все об этом файле.

Формат, содержание и размещение индексных дескрипторов несколько различаются в разных системах (особенно когда идет речь о сети), но следующие характеристики присущи практически каждому дескриптору:

1. Тип файла, 9 битов защиты RWX и некоторые другие биты.
2. Число связей с файлом (число элементов директорий).
3. Идентификатор владельца.
4. Группа владельца.
5. Длина файла в байтах.
6. Тринадцать адресов на диске.
7. Время, когда файл читали в последний раз.
8. Время, когда последний раз производилась запись в файл.
9. Время, когда в последний раз менялся индексный дескриптор.

Типы файлов бывают следующие: обычные файл, директории и два вида особых файлов для устройств ввода-вывода с блочной структурой и неструктурированных устройств ввода-вывода соответственно. Число связей и идентификатор владельца мы уже обсуждали. Длина файла выражается 32-битным целым числом, которое показывает самый старший байт файла. Вполне возможно создать файл, перенести указатель на позицию 1 000 000 и записать 1 байт. В результате получится файл длиной 1 000 001. Тем не менее этот файл не требует сохранения всех отсутствующих байтов.

Первые 10 адресов на диске указывают на блоки данных. Если размер блока — 1024 байта, то можно работать с файлами размером до 10 240 байтов. Адрес 11 указывает на **блок косвенной адресации**, который содержит 256 адресов диска. Здесь можно работать с файлами размером до 10 240+256×1024=272 384 байта. Для файлов еще большего размера существует адрес 12, который указывает на 256 блоков косвенной адресации. Здесь допустимый размер файлов составляет 272384+256×256×1024=67 381 248 байтов. Если и эта схема **блока двойной косвенной адресации** слишком мала, то используется адрес 13. Он указывает на **блок тройной косвенной адресации**, который содержит адреса 256 блоков двойной косвенной адресации. Используя прямую, косвенную, двойную косвенную и тройную косвенную адресацию, можно обращаться к 16 843 018 блокам. Это значит, что максимально возможный размер файла составляет 17 247 250 432 байта. Поскольку размер указателей файлов ограничен до 32 битов, реальный верхний предел на размер файла составляет 4 294 967 295 байтов. Свободные блоки диска хранятся в связном списке. Если нужен новый блок, он берется из списка. В результате получается, что блоки каждого файла беспорядочно раскиданы по всему диску.

Чтобы повысить скорость ввода-вывода с диска, нужно сделать следующее. После открытия файла его индексный дескриптор копируется в таблицу в основной памяти и хранится там, пока файл остается открытым. Кроме того, в памяти находится набор блоков, к которым недавно производилось обращение. Так как большинство файлов считывается последовательно, часто при обращении к файлу требуется тот же блок, что и при предыдущем обращении. Чтобы увеличить скорость, система считывает следущий блок в файл еще до того, как к нему произведено обращение. Все эти моменты скрыты от пользователя. Когда пользователь выдает вызов *read*, работа программы приостанавливается, пока требуемые данные не появятся в буфере.

Зная все это, мы теперь можем рассмотреть, как происходит процесс ввода-вывода. *Open* заставляет систему искать директории по определенному пути. Если поиск успешен, то индексный дескриптор считывается во внутреннюю таблицу. Вызовы *read* и *write* требуют, чтобы система вычислила номер блока из текущей позиции файла. Адреса первых 10 блоков диска всегда находятся в основной памяти (в индексном дескрипторе); для остальных блоков сначала требуется считать один или несколько блоков косвенной адресации. *Lseek* просто меняет текущую позицию указателя и не производит никакого ввода-вывода.

Link смотрит на свой первый аргумент, чтобы обнаружить номер индексного дескриптора. Затем он создает элемент директории для второго аргумента и помещает номер индексного дескриптора первого файла в этот элемент директории. Наконец, он увеличивает число связей в индексном дескрипторе на 1. *Unlink* удаляет элемент директории и уменьшает число связей в индексном дескрипторе. Если это число равно 0, файл удаляется и все блоки помещаются в список свободных блоков.

Виртуальный ввод-вывод в Windows NT

NT поддерживает несколько файловых систем, самые важные из которых — **NTFS (NT File Sysytem — файловая система Windows NT)** и **FAT (File Allocation Table — таблица размещения файлов)**. Первая была разработана специально для NT. Вторая является старой файловой системой для MS-DOS, которая также используется в Windows 95/98 (хотя и с длинными именами файлов). Поскольку система FAT устарела, ниже мы рассмотрим только файловую систему NTFS. FAT32 начала использоваться с NT 5.0. Она подерживалась и в более позних версиях Windows 95 и Windows 98.

В файловой системе NT длина имени файла может быть до 255 символов. Имена файлов написаны в коде Unicode, благодаря чему люди в разных странах, где не используется латинский алфавит, могут писать имена файлов на их родном языке. В файловой системе NT заглавные и строчные буквы в именах файлов считаются разными (то есть foo отличается от FOO). К сожалению, в системе Win32 API заглавные и строчные буквы в именах файлов и директорий не различаются, поэтому это преимущество теряется для программ, которые используют Win32.

Как и в системе UNIX, файл представляет собой линейную последовательность байтов, максимальная длина $2^{64}-1$. Указатели тоже существуют, но их длина не 32, а 64 бита, чтобы можно было поддерживать максимальную длину файла. Вызовы функций в Win32 API для манипуляций с директориями и файлами в целом схожи с вызовами функций в системе UNIX, но большинство из них имеют больше

параметров, и модель защиты другая. При открытии файла возвращается идентификатор (handle), который затем используется для чтения и записи файла. В отличие от системы UNIX, идентификаторы не являются маленькими целыми числами, а стандартный ввод, стандартный вывод и стандартная ошибка не определяются заранее как 0, 1 и 2 (исключение составляет пультовый режим работы). Основные функции Win32 API для управления файлами приведены в табл. 6.11.

Таблица 6.11. Основные функции Win32 API для ввода-вывода файлов. Во второй колонке дается эквивалент из UNIX

Функция API	UNIX	Значение
CreateFile	open	Создает файл или открывает существующий файл; возвращает идентификатор
DeleteFile	unlink	Удаляет существующий файл
CloseHandle	close	Закрывает файл
ReadFile	read	Считывает данные из файла
WriteFile	write	Записывает данные в файл
SetFilePointer	lseek	Устанавливает указатель файла на определенное место в файле
SetFileAttributes	stat	Возвращает свойства файла
LockFile	Fcntl	Блокирует область файла, чтобы обеспечить взаимное исключение доступа
UnlockFile	Fcntl	Снимает блокировку с ранее заблокированной области файла

Рассмотрим эти вызовы. *CreateFile* используется для создания нового файла и возвращает идентификатор (handle) для него. Эта функция применяется и для открытия уже существующего файла, поскольку в системе API нет функции *open*. Мы не будем приводить параметры функций API, поскольку их очень много. Например, *CreateFile* имеет семь параметров:

1. Указатель на имя файла, который нужно создать или открыть.
2. Флаги, которые сообщают, какие действия разрешено производить с файлом: читать, записывать или и то и другое.
3. Флаги, которые сообщают, могут ли несколько процессов открывать файл одновременно.
4. Указатель на дескриптор безопасности, который сообщает, кто имеет доступ к файлу.
5. Флаги, которые сообщают, что нужно делать, если файл существует или не существует.
6. Флаги, связанные с атрибутами архивации, компрессии и т. д.
7. Идентификатор файла (handle), атрибуты которого нужно клонировать для нового файла.

Следующие шесть функций API сходны с соотвествующими функциями в системе UNIX. Последние две позволяют блокировать и разблокировать область файла, чтобы обеспечить взаимное исключение доступа.

Используя эти функции API, можно написать процедуру для копирования файла, аналогичную процедуре в листинге 6.3. Такая процедура (без проверки оши-

бок) приведена в листинге 6.4. На практике программу для копирования файла писать не нужно, поскольку существует функция *CopyFile*.

Листинг 6.4. Фрагмент программы для копирования файла с применением функции API из системы Windows NT. Фрагмент написан на языке C, язык Java не показывает системные вызовы низкого уровня, а нам нужно было продемонстрировать их

```
/* Открытие файлов для ввода и вывода. */
inhandle = CreateFile("data", GENERIC_READ, 0, NULL,     OPEN_EXISTING, 0, NULL);
outhandle = CreateFileC'newF, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL,
NULL);
/* Копирование файла. */
do{
       s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);
       if (s > 0 && count > 0) WriteFile(outhandle, buffer, count, &ocnt, NULL);
while (s > 0 && count > 0);
/* Закрытие файлов. */
CloseHandle(inhandle);
CloseHandle(outhandle);
```

NT поддерживает иерархическую систему файлов, сходную с системой файлов UNIX. Однако в качестве разделителя здесь используется не /, а \ (заимствовано из MS-DOS). Здесь тоже существует понятие текущего каталога, а пути могут быть абсолютными и относительными. Однако между NT и UNIX есть одно существенное различие. UNIX позволяет монтировать в одно дерево системы файлов с разных дисков и машин, скрывая таким образом структуру диска от программного обеспечения. NT 4.0 не имеет этого качества, поэтому абсолютные имена файлов должны начинаться с буквы, которая указывает на диск (например, C:\windows\system\foo.dll). Свойство монтирования систем файлов появилось с NT 5.0.

Основные функции для работы с директориями приведены в табл. 6.12 (также вместе с эквивалентами из UNIX). Думаем, что раскрывать их значение не требуется.

Отметим, что NT 4.0 не поддерживает связи файлов. На уровне графического рабочего стола поддерживаются клавишные комбинации быстрого вызова, но эти структуры не имели соответствий в самой системе файлов. Прикладная программа не могла войти в файл во второй директории, не скопировав весь файл. Начиная с NT 5.0 к системе файлов были добавлены файловые связи.

Таблица 6.12. Основные функции Min32 API для работы с директориями. Во втором столбце даны эквиваленты из UNIX, если они существуют

Функция API	UNIX	Значение
CreateDirectiry	mkdir	Создает новую директорию
RemoveDirectory	rmdir	Удаляет пустую директорию
FindFirstFile	opendir	Инициализирует чтение элементов директории
FindNextFile	readdir	Читает следующий элемент директории
MoveFile		Перемещает файл из одной директории в другую
SetCurrentDirectory	chdir	Меняет текущую директорию

NT имеет более сложный механизм защиты, чем в UNIX. Когда пользователь входит в систему, его процесс получает **маркер доступа** от операционной системы.

Маркер доступа содержит **идентификатор безопасности (SID – Security ID)**, список групп, к которым принадлежит пользователь, имеющиеся привилегии и некоторую другую информацию. Маркер доступа концентрирует всю информацию о защите в одном легко доступном месте. Все процессы, созданные этим процессом, наследуют этот же маркер доступа.

Дескриптор защиты – это один из параметров, который дается при создании любого объекта. Дескриптор защиты содержит список элементов, который называется **списком контроля доступа (ACL – Access Control List).** Каждый элемент разрешает или запрещает совершать определенный набор операций над объектом какому-либо отдельному человеку или группе. Например, файл может содержать дескриптор защиты, который определяет, что Иванов не имеет доступа к файлу вообще, Петров может читать файл, Сидоров может читать и записывать файл, а все члены группы XYZ могут прочитать только размер файла.

Если процесс пытается выполнить какую-либо операцию над объектом с использованием идентификатора (handle), который он получил при открытии объекта, диспетчер безопасности получает маркер доступа данного процесса и начинает перебирать элементы списка контроля доступа по порядку. Как только он находит элемент, который соответствует нужному пользователю или одной из групп, информация о разрешении или запрещении доступа, найденная там, принимается в качестве заданной. По этой причине элементы, запрещающие доступ, обычно помещаются в список контроля доступа перед элементами, разрешающими доступ (чтобы пользователь, у которого нет доступа, не смог получить его незаконно, будучи членом одной из групп, которой доступ разрешен). Дескриптор защиты также содержит информацию, используемую для аудита доступов к объекту.

А теперь рассмотрим, как файлы и директории реализуются в NT. Каждый диск разделен на тома, такие же как разделы диска в UNIX. Каждый том содержит файлы, битовые отображения директорий и другие структуры данных. Каждый том организован в виде линейной последовательности **кластеров**. Размер кластера фиксирован для каждого тома. Он может быть от 512 байтов до 64 Кбайт, в зависимости от размера тома. Обращение к кластеру осуществляется по смещению от начала тома. При этом используются 64-битные числа.

Основной структурой данных в каждом томе является **MFT (Master File Table – главная файловая таблица)**, в которой содержится элемент для каждого файла и директории в томе. Эти элементы аналогичны элементам индексного дескриптора (i-node) в системе UNIX. Главная файловая таблица является файлом и может быть помещена в любое место в пределах тома. Это устраняет проблему, возникающую при наличии испорченных блоков на диске в середине индексных дескрипторов.

Главная файловая таблица показана на рис. 6.28. Она начинается с заголовка, в котором дается информация о томе (указатели на корневой каталог, файл загрузки, список лиц, пользующихся свободным доступом и т. д.). Затем идет по одному элементу на каждый файл или директорию (1 Кбайт за исключением тех случаев, когда размер кластера составляет 2 Кбайт и более). Каждый элемент содержит все метаданные (административную информацию) о файле или директории. Допускается несколько форматов, один из которых изображен на рис. 6.28.

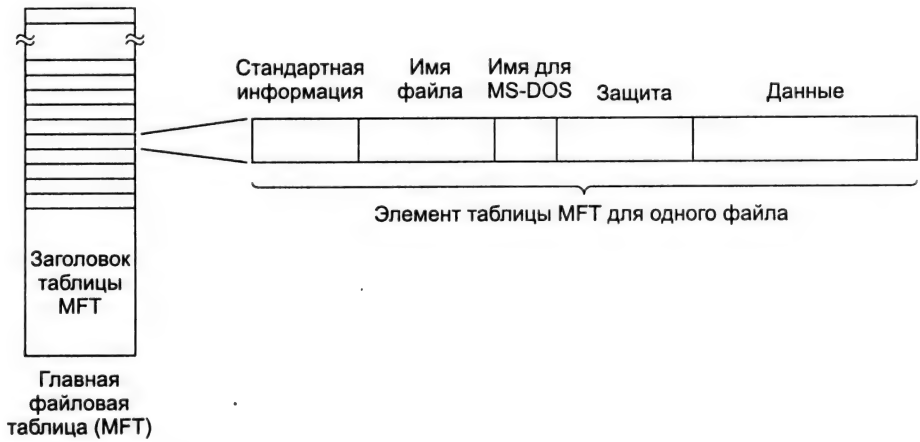


Рис. 6.28. Главная файловая таблица в системе Windows NT

Поле стандартой информации содержит информацию об отметках времени, необходимых стандартах POSIX, о числе связей, о битах «только для чтения» и битах архивирования и т. д. Это поле имеет фиксированную длину и является обязательным. Имя файла может иметь любую длину до 255 символов Unicode. Чтобы такие файлы стали доступны для старых 16-битных программ, они могут снабжаться дополнительным именем MS-DOS, состоящим максимум из 8 символов, за которыми следует точка и расширение из 3 символов. Если действительное имя файла соответствует правилу наименования в MS-DOS (8+3), то второе имя для MS-DOS не используется.

Далее следует информация о защите. Во всех версиях, вплоть до NT 4.0, в поле защиты содержался дескриптор защиты. Начиная с NT 5.0, вся информация о защите помещается в один файл, а поле защиты просто указывает на соответствующую часть этого файла.

Для файлов небольшого размера сами данные этих файлов содержатся в элементе главной файловой таблицы, что упрощает их вызов, — для этого не требуется обращаться к диску. Данная идея получила название **непосредственный файл** (immediate file) [100]. Для файлов большого размера это поле содержит указатели на кластеры, в которых содержатся данные или (что более распространено) блоки последовательных кластеров, так что номер кластера и его длина могут представлять произвольное количество данных. Если элемент главной файловой таблицы недостаточно велик для хранения нужной информации, к нему можно привязать один или несколько дополнительных элементов.

Максимальный размер файла составляет 2^{64} байтов. Поясним, что собой представляет файл на 2^{64} байтов. Представим, что файл был написан в двоичной системе, а каждый 0 или 1 занимает 1 мм пространства. Длина листинга на 2^{67} мм составила бы 15 световых лет. Этого хватило бы для того, чтобы выйти за пределы Солнечной системы, достичь Альфа Центавра и вернуться обратно.

Файловая система NTFS имеет много других интересных особенностей, в том числе возможность компрессии данных и отказоустойчивость с применением атомарных транзакций. Дополнительную информацию об этой системе можно найти в [137].

Примеры управления процессами

Системы NT и UNIX позволяют разделить работу на несколько процессов, которые выполняются параллельно и взаимодействуют друг с другом, как в примере с производителем и потребителем, который мы обсуждали ранее. В этом разделе мы поговорим о том, как происходит управление процессами в обеих системах. Обе системы поддерживают параллелизм в пределах одного процесса с использованием потоков, и об этом мы тоже расскажем.

Управление процессами в системе UNIX

В любой момент процесс в системе UNIX может создать подпроцесс, который является его точной копией. Для этого выполняется системный вызов *fork*. Изначальный процесс называется **порождающим процессом**, а новый — **порожденным процессом**. Два процесса, полученные в результате выполнения операции *fork*, абсолютно идентичны и даже разделяют одни и те же файловые дескрипторы. Каждый из этих двух процессов выполняет свою работу независимо от другого.

Часто порожденный процесс определенным образом дезориентирует дескрипторы файлов, а затем выполняет системный вызов *exec*, который замещает его программу и данные программой и данными из выполняемого файла, определенного в качестве параметра к вызову *exec*. Например, если пользователь печатает команду `xyz`, то интерпретатор команд (оболочка) выполняет операцию *fork*, создавая таким образом порожденный процесс. А этот процесс выполняет процедуру *exec*, чтобы запустить программу *xyz*.

Эти два процесса работают параллельно, но иногда порождающий процесс должен по каким-либо причинам ждать, чтобы порожденный процесс завершил свою работу, и только после этого продолжать выполнение тех или иных действий. В этом случае порождающий процесс выполняет системный вызов *wait* или *waitpid*, в результате чего он временно приостанавливается и ждет, пока порожденный процесс не выполнит системный вызов *exit*.

Процессы могут выполнять процедуру *fork* сколь угодно часто, в результате чего получается целое дерево процессов. Посмотрите на рис. 6.29. Здесь процесс A выполнил процедуру *fork* дважды и породил два новых процесса, B и C. Затем процесс B тоже выполнил процедуру *fork* дважды, а процесс C выполнил ее один раз. Таким образом, получилось дерево из шести процессов.

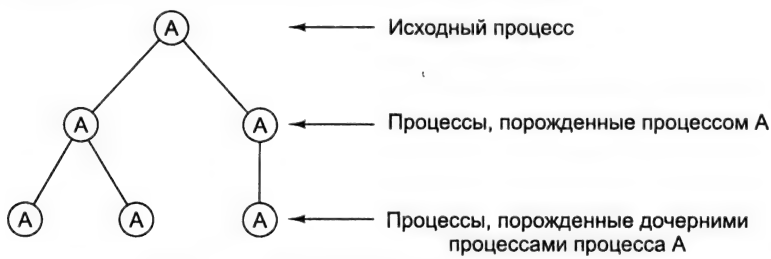


Рис. 6.29. Дерево процессов в системе UNIX

Процессы в системе UNIX могут взаимодействовать друг с другом через специальную информационную структуру, которая называется **каналом**. Канал пред-

ставляет собой вид буфера, в который один процесс записывает поток данных, а другой процесс забирает эти данные оттуда. Байты всегда возвращаются из канала в том порядке, в котором они были записаны. Случайный доступ невозможен. Каналы не сохраняют границы между отрезками данных, поэтому если один процесс записал в канал 4 отрезка по 128 байтов, а другой процесс считывает данные по 512 байтов, то этот второй процесс получит все данные сразу без указания на то, что они были записаны в несколько заходов.

В системах System V и Solaris применяется другой метод взаимодействия процессов. Здесь используются так называемые **очереди сообщений**. Процесс может создать новую очередь сообщений или открыть уже существующую с помощью вызова *msgget*. Для отправки сообщений используется вызов *msgsnd*, а для получения — *msgrecv*. Сообщения, отправленные таким способом, отличаются от данных, помещаемых в канал. Во-первых, границы сообщений сохраняются, а в канал передается просто поток данных. Во-вторых, сообщения имеют приоритеты, поэтому срочные сообщения идут перед всеми остальными. В-третьих, сообщения типизированы, и вызов *msgrecv* может определять их тип, если это необходимо.

Два и более процесса могут разделять общую область своих адресных пространств. UNIX управляет этой разделенной памятью путем отображения одних и тех же страниц в виртуальное адресное пространство всех разделенных процессов. В результате запись в общую область, произведенная одним из процессов, будет видна всем остальным процессам. Этот механизм обеспечивает очень высокую пропускную способность между процессами. Системные вызовы, включенные в разделенную память, идут по алфавиту (как *shmat* и *shmop*).

Еще одна особенность System V и Solaris — наличие семафоров. Принципы их работы мы уже описывали, когда говорили о производителе и потребителе.

Системы UNIX могут поддерживать несколько потоков управления в пределах одного процесса. Эти потоки управления обычно называют просто **потоками**. Они похожи на процессы, которые делят общее адресное пространство и все объекты, связанные с этим адресным пространством (дескрипторы файлов, переменные окружения и т. д.). Однако каждый поток имеет свой собственный счетчик команд, свои собственные регистры и свой собственный стек. Если один из потоков приостанавливается (например, пока не завершится процесс ввода-вывода), другие потоки в том же процессе могут продолжать работу. Два потока в одном процессе, которые действуют как процесс-производитель и процесс-потребитель, сходны с двумя однопоточными процессами, которые разделяют сегмент памяти, содержащий буфер, хотя не идентичны им. Во втором случае каждый процесс имеет свои собственные дескрипторы файлов и т. д., тогда как в первом случае все эти элементы общие.

В каких случаях могут понадобиться потоки? Рассмотрим сервер World Wide Web. Такой сервер может хранить в основной памяти кэш часто используемых web-страниц. Если нужная страница находится в кэш-памяти, то она выдается немедленно. Если нет, то она вызывается с диска. К сожалению, на это требуется довольно длительное время (обычно 20 млс), и на это время процесс блокируется и не может обслуживать новые поступающие запросы, даже если эти web-страницы находятся в кэш-памяти.

По этой причине было принято решение сделать несколько потоков в одном процессе, которые разделяют общую кэш-память web-страниц. Если один из потоков блокируется, новые запросы могут обрабатываться другими потоками. Предотвратить блокировку процессов можно и без использования потоков. Для этого потребуется иметь несколько процессов, но тогда нужно будет продублировать кэш, а это несколько расточительно, поскольку размер памяти ограничен.

Стандарт системы UNIX для потоков называется **pthreads**. Он определяется стандартом POSIX (P1003.1C) и содержит вызовы для управления потоками и их синхронизации. Управляется ли потоками ядро, или они полностью находятся в пользовательском пространстве, в стандарте не определено. Наиболее распространенные вызовы для работы с потоками приведены в табл. 6.13.

Таблица 6.13. Основные вызовы для потоков, определенные в стандарте POSIX

Вызов потока	Значение
pthread_create	Создает новый поток в адресном пространстве вызывающей процедуры
pthread_exit	Завершает поток
pthread_join	Ждет завершения потока
pthread_mutex_init	Создает новый мьютекс
pthread_mutex_destroy	Удаляет мьютекс
pthread_mutex_lock	Блокирует мьютекс
pthread_mutex_unlock	Снимает блокировку с мьютекса
pthread_cond_init	Создает переменную условия
pthread_cond_destroy	Удаляет переменную условия
pthread_cond_wait	Ждет переменную условия
pthread_cond_signal	Снимает блокировку с одного из потоков, который ждет переменную условия

Давайте рассмотрим эти вызовы. Первый вызов, *pthread_create*, создает новый поток. После выполнения этой процедуры в адресном пространстве появляется на один поток больше. Поток, который выполнил свою работу, вызывает *pthread_exit*. Если потоку нужно подождать, пока другой поток не окончит свою работу, он вызывает *pthread_join*. Если этот другой поток уже закончил свою работу, вызов *pthread_join* немедленно завершается.

Потоки можно синхронизировать с помощью специальных объектов, которые называются **мьютексами**. Обычно мьютекс управляет каким-либо ресурсом (например, буфером, разделенным между двумя потоками). Для того чтобы в конкретный момент времени только один поток мог получать доступ к общему ресурсу, потоки дожны запирать мьютекс перед использованием ресурса и отпирать его после завершения работы с ним. Таким образом можно избежать состояния гонок, поскольку этому протоколу подчиняются все потоки. Мьютексы похожи на бинарные семафоры (то есть семафоры, которые могут принимать только два значения: 0 или 1). Эти объекты получили название «мьютексы» (mutexes), поскольку они используются для обеспечения взаимного исключения доступа к какому-либо из ресурсов (MUTual EXclusion по-английски значит «взаимное исключение»).

Мьютексы можно создавать и разрушать с помощью вызовов *pthread_mutex_init* и *pthread_mutex_destroy* соответственно. Мьютекс может находиться в одном из двух состояний: блокированном и неблокированном. Если потоку нужно установить блокировку на незапертый мьютекс, он использует *pthread_mutex_lock*, а затем продолжает работу. Однако если поток пытается запереть мьютекс, который уже заперт, он блокируется. Когда поток, который в данный момент использует общий ресурс, завершит работу с этим ресурсом, он должен разблокировать соответствующий мьютекс с помощью *pthread_mutex_unlock*.

Мьютексы предназначены для кратковременной блокировки (например, для защиты общей переменной). Они не предназначены для длительной синхронизации (например, для ожидания, пока освободится накопитель на магнитной ленте). Для длительной синхронизации существуют **переменные условия (condition variables)**. Эти переменные создаются и удаляются с помощью вызовов *pthread_cond_init* и *pthread_cond_destroy* соответственно.

Если, например, поток обнаружил, что накопитель на магнитной ленте, который ему нужен, в данный момент занят, этот поток совершает *pthread_cond_wait* над переменной условия. Когда поток, который использует накопитель на магнитной ленте, завершает свою работу с этим устройством (а это может произойти через несколько часов), он посылает сигнал *pthread_cond_signal*, чтобы разблокировать ровно один поток, который ожидает эту переменную условия. Если ни один поток не ожидает эту переменную, сигнал пропадает. У переменных условия (condition variables) нет счетчика, как у семафоров. Отметим, что над потоками, мьютексами и переменными условия можно выполнять и некоторые другие операции.

Управление процессами в Windows NT

NT поддерживает несколько процессов, которые могут взаимодействовать и синхронизироваться. Каждый процесс содержит по крайней мере один поток, который, в свою очередь, содержит по крайней мере одну **нить** (fiber) (это легковесный поток). Процессы, потоки и нити в совокупности обеспечивают набор средств для поддержания параллелизма и в машинах с одним процессором, и в многопроцессорных системах.

Новые процессы создаются с помощью функции API *CreateProcess*. Эта функция имеет 10 параметров, каждый из которых имеет множество опций. Ясно, что такая разработка гораздо сложнее соответствующей схемы в UNIX, где *fork* вообще не имеет параметров, а *exec* имеет всего три параметра: указатели на имя файла, который нужно выполнить, на массив параметров командной строки и на строки описания конфигурации. Ниже изложены 10 параметров функции *CreateProcess*:

1. Указатель на имя выполняемого файла.
2. Сама командная строка.
3. Указатель на дескриптор защиты для данного процесса.
4. Указатель на дескриптор защиты для внутреннего потока.
5. Бит, который сообщает, наследует ли новый процесс идентификаторы (handles) исходного процесса.
6. Различные флаги (например, ошибка, приоритет, отладка, консоль).
7. Указатель на строки описания конфигурации.

8. Указатель на имя текущего каталога нового процесса.
9. Указатель на структуру, которая описывает исходное окно экрана.
10. Указатель на структуру, которая возвращает 18 значений вызывающей процедуре.

В NT нет никакой иерархии типа порождающий—порожденный. Все процессы создаются равными. Но поскольку одним из 18 параметров, возвращаемых исходному процессу, является идентификатор (handle) для нового процесса (а это дает возможность контролировать новый процесс), здесь существует внутренняя иерархия с точки зрения того, что определенные процессы содержат идентификаторы других процессов. Эти идентификаторы нельзя просто непосредственно передавать другим процессам, но процесс может сделать определенный идентификатор доступным для другого процесса, а затем передать ему этот идентификатор, так что внутренняя иерархия процессов не может сохраняться долго.

Каждый процесс в NT создается с одним потоком, но позднее этот процесс может создать еще несколько потоков. Создание потока проще, чем создание процесса, поскольку *CreateThread* имеет всего 6 параметров вместо 10: дескриптор защиты, размер стека, начальный адрес, определяемый пользователем параметр, начальное состояние потока (готов к работе или блокирован) и идентификатор потока.

Создание потоков производится ядром (то есть потоки реализуются не в пользовательском пространстве, как в некоторых других системах).

Когда ядро совершает распределение, оно вызывает не только процесс, который должен запускаться следующим, но и поток в этом процессе. Это значит, что ядро всегда знает, какие потоки блокированы, а какие — нет. Так как потоки являются объектами ядра, они имеют дескрипторы защиты и идентификаторы. Поскольку идентификатор разрешается передавать другому процессу, можно сделать так, чтобы один процесс управлял потоками в другом процессе. Эта особенность может понадобиться для программ отладки.

Создавать потоки в NT довольно расточительно, поскольку для создания потока требуется войти в ядро, а затем выйти из него. Чтобы избежать этого, в NT предусмотрены **нити** (fibers), которые похожи на потоки, но распределяются в пользовательском пространстве программой, которая их создает. Каждый поток может иметь несколько нитей, точно так же как процесс может иметь несколько потоков, только в данном случае, когда нить блокируется, она встает в очередь заблокированных нитей и выбирает другую нить для работы в своем потоке. Ядро не знает об этом переходе, поскольку поток все равно продолжает работать, даже если сначала действовала одна нить, а затем другая. Ядро управляет процессами и потоками, но не управляет нитями. Нити могут пригодиться, например, в том случае, когда программы, которые управляют своими собственными потоками, переносятся в NT.

Процессы могут взаимодействовать друг с другом разными способами: через каналы, именованные каналы, почтовые ящики, сокеты (sockets), удаленные вызовы процедур и общие файлы. Каналы бывают двух видов: байтовые каналы и каналы сообщений. Тип выбирается во время создания. Байтовые каналы работают так же, как в системе UNIX. Каналы сообщений сохраняют границы сообщений, поэтому четыре записи по 128 байтов будут прочитаны как четыре сообщения по 128 байтов (а не как одно сообщение на 512 байтов, как в случае с байтовыми каналами). Кроме того, существуют именованные каналы, которые тоже бывают двух видов. Именованные каналы могут использоваться в сети, а обычные каналы — нет.

Почтовые ящики есть только в NT (в системе UNIX их нет). Они во многом похожи на каналы, хотя не во всем. Во-первых, они односторонние, а каналы — двусторонние. Их можно использовать в сети, но они не гарантируют доставку. Наконец, они позволяют отправлять сообщение нескольким получателям, а не только одному.

Сокеты похожи на каналы, но они обычно связывают процессы на разных машинах. Однако их можно применять и для связи процессов на одной машине. Вообще говоря, связь через сокет ненамного выгоднее связи через обычный или именованный канал.

Удаленные вызовы процедур позволяют процессу A приказать процессу B совершить вызов процедуры в адресном пространстве B от имени A и возвратить результат процессу A. Здесь существуют различные ограничения на параметры. Например, не имеет никакого смысла передача указателя другому процессу.

Наконец, процессы могут разделять общую память путем отображения одновременно в один и тот же файл. Тогда все записи, произведенные одним процессом, появляются в адресном пространстве других процессов. Применяя такой механизм, можно легко реализовать разделенный буфер, который мы описывали в примере с процессом-производителем и процессом-потребителем.

NT предоставляет множество механизмов синхронизации (семафоры, мьютексы, критические секции и события). Все эти механизмы работают с потоками, но не с процессами, поэтому когда поток блокируется на семафоре, это никак не влияет на другие потоки в этом процессе — они просто продолжают работать.

Семафор создается с помощью функции API *CreateSemaphore*, которая может установить его на определенное значение и определить его максимальное значение. Семафоры являются объектами ядра, поэтому они имеют дескрипторы защиты и идентификаторы (handles). Идентификатор для семафора может дублироваться с помощью *DuplicateHandle* и передаваться другому процессу, поэтому на одном семафоре можно синхронизировать несколько процессов. Присутствуют функции *up* и *down*, хотя они имеют особые названия: *ReleaseSemaphore* (*up*) и *WaitForSingleObject* (*down*). Можно определить для функции *WaitForSingleObject* предел на время простоя, и тогда вызывающий поток в конце концов может быть разблокирован, даже если семафор сохраняет значение 0.

Мьютексы тоже являются объектами ядра, но они проще семафоров, поскольку у них нет счетчиков. Они, по сути, представляют собой объекты с функциями API для блокирования (*WaitForSingleObject*) и разблокирования (*ReleaseMutex*). Идентификаторы мьютексов, как и идентификаторы семафоров, можно дублировать и передавать другим процессам, так что потоки из разных процессов могут иметь доступ к одному и тому же мьютексу.

Третий механизм синхронизации основан на **критических секциях**, которые сходны с мьютексами, за исключением локальности по отношению к адресному пространству исходного потока. Поскольку критические секции не являются объектами ядра, у них нет идентификаторов (handles) и дескрипторов защиты и их нельзя передавать другим процессам. Блокировка и разблокировка осуществляются с помощью *EnterCriticalSection* и *LeaveCriticalSection* соответственно. Так как эти функции API выполняются целиком в пользовательском пространстве, они работают гораздо быстрее, чем мьютексы.

Последний механизм связан с использованием объектов ядра, которые называются **событиями**. Если потоку нужно дождаться какого-то события, он вызывает *WaitForSingleObject*. Можно разблокировать один ожидающий поток с помощью *SetEvent* или все ожидающие потоки с помощью *PulseEvent*. Существует несколько видов событий, которые имеют несколько опций.

События, мьютексы и семафоры можно назвать определенным образом и хранить в системе файлов как именованные каналы. Можно синхронизировать работу двух и более процессов путем открытия одного и того же события, мьютекса или семафора. В этом случае им не нужно создавать объект, а затем дублировать идентификаторы, хотя такой подход тоже возможен.

Краткое содержание главы

Операционную систему можно считать интерпретатором определенных особенностей архитектуры, которых нет на уровне команд. Главными среди них являются виртуальная память, виртуальные команды ввода-вывода и средства параллельной обработки.

Виртуальная память нужна для того, чтобы позволить прграммам использовать больше адресного пространства, чем есть у машины на самом деле, или чтобы обеспечить удобный механизм для защиты и разделения памяти. Виртуальную память можно реализовать в виде чистого разбиения на страницы, чистой сегментации или того и другого вместе. При страничной организации памяти адресное пространство разбивается на равные по размеру виртуальные страницы. Одни из них отображаются на физические страничные кадры, другие — нет. Отсылка к отображенной странице преобразуется контроллером управления памятью в правильный физический адрес. Обращение к неотображенной странице вызывает ошибку из-за отсутствия страницы. Pentium II и UltraSPARC II имеют сложные контроллеры управления памятью, которые поддерживают виртуальную память и страничную организацию.

Самой важной абстракцией ввода-вывода на этом уровне является файл. Файл состоит из последовательности байтов или логических записей, которые можно читать и записывать, не зная при этом о том, как работают диски и другие устройства ввода-вывода. Доступ к файлам может осуществляться последовательно, непоследовательно по номеру записи и непоследовательно по ключу. Для группировки файлов используются директории. Файлы могут храниться в последовательных секторах, а могут быть разбросаны по диску. В последнем случае требуются специальные структуры данных для нахождения всех блоков файла. Чтобы следить за свободным пространством на диске, можно использовать список или битовое отображение.

В системах часто поддерживается параллельная обработка. Для этого путем разделения времени в одном процессоре моделируется несколько процессов. Неконтролируемое взаимодействие различных процессов может привести к состоянию гонок. Чтобы избежать их, вводятся специальные средства синхронизации. Самыми простыми из них являются семафоры.

UNIX и NT являются сложными операционными системами. Обе системы поддерживают страничную организацию памяти и отображаемые в память файлы.

Кроме того, они поддерживают иерархические системы файлов, в которых файлы состоят из последовательности байтов. Наконец, обе системы поддерживают процессы и потоки и обеспечивают механизмы их синхронизации.

Вопросы и задания

1. Почему операционная система интепретирует только некоторые команды третьего уровня, тогда как микропрограмма интерпретирует все команды этого уровня (уровня архитектуры команд)?
2. Машина содержит 32-битное виртуальное адресное пространство с побайтовой адресацией. Размер страницы составляет 8 Кбайт. Сколько существует страниц виртуального адресного пространства?
3. Виртуальная память содержит 8 виртуальных страниц и 4 физических страничных кадра. Размер страницы составляет 1024 слова. Ниже приведена таблица страниц:

Виртуальная страница	Страничный кадр
0	3
1	1
2	Нет в основной памяти
3	Нет в основной памяти
4	2
5	Нет в основной памяти
6	0
7	Нет в основной памяти

   1. Создайте список виртуальных адресов, обращение к которым будет вызывать ошибку из-за отсутствия страницы.
   2. Каковы физические адреса для 0, 3728, 1023, 1024, 1025, 7800 и 4096?
4. Компьютер имеет 16 страниц виртуального адресного пространства и только 4 страничных кадра. Изначально память пуста. Программа обращается к виртуальным страницам в следующем порядке:

   0, 7, 2, 7, 5, 8, 9, 2, 4

   а. Какие из обращений вызовут ошибку с алгоритмом LRU?

   б. Какие из обращений вызовут ошибку с алгоритмом FIFO?
5. В разделе «Политика замещения страниц» был предложен алогоритм замещения страниц FIFO. Разработайте более эффективный алгоритм. *Подсказка*: можно обновлять счетчик во вновь загружаемой странице, оставляя все другие.
6. В системах со страничной организацией памяти, которые мы обсуждали в этой главе, обработчик ошибок, происходящих из-за отсутствия страниц, был частью уровня архитектуры команд и, следовательно, не присутствовал в адресном пространстве операционной системы. На практике же этот обработчик

занимает некоторые страницы и может быть удален при определенных обстоятельствах (например, в соответствии с политикой замещения страниц). Что бы случилось, если бы обработчика ошибок не было в наличии в тот момент, когда произошла ошибка? Как можно было бы разрешить эту проблему?

7. Не все компьютеры содержат специальный бит, который автоматически устанавливается, когда производится запись в страницу. Однако нужно каким-то образом следить, какие страницы были изменены, чтобы не пришлось записывать все страницы обратно на диск после их использования. Если предположить, что каждая страница имеет специальные биты для разрешения чтения, записи и выполнения, то как операционная система может следить, какие страницы изменялись, а какие — нет?
8. Сегментированная память содержит страничные сегменты. Каждый виртуальный адрес содержит 2-битный номер сегмента, 2-битный номер страницы и 11-битное смещение внутри страницы. Основная память содержит 32 Кбайт, которые разделены на страницы по 2 Кбайт. Каждый сегмент разрешается либо только читать, либо читать и выполнять, либо читать и записывать, либо читать, записывать и выполнять. Таблицы страниц с указанием на защиту приведены ниже:

Сегмент 0		**Сегмент 1**	**Сегмент 2**	**Сегмент 3**		
Только для чтения		Чтение/выполнение		Чтение/запись/ выполнение	Чтение/запись	
Виртуальная страница	Страничный кадр	Виртуальная страница	Страничный кадр		Виртуальная страница	Страничный кадр
0	9	0	На диске	Таблицы	0	14
1	3	1	0	страниц нет	1	1
2	На диске	2	15	в основной	2	6
3	12	3	8	памяти	3	На диске

Вычислите физический адрес для каждого из нижеперечисленных доступов к виртуальной памяти. Если происходит ошибка, скажите, какого она типа.

Доступ	Сегмент	Страница	Смещение внутри страницы
1. Вызов данных	0	1	1
2. Вызов данных	1	1	10
3. Вызов данных	3	3	2047
4. Сохранение данных	0	1	4
5. Сохранение данных	3	1	2
6. Сохранени данных	3	0	14
7. Переход	1	3	100
8. Вызов данных	0	2	50
9. Вызов данных	2	0	5
10. Переход	3	0	60

9. Некоторые компьютеры позволяют осуществлять ввод-вывод непосредственно в пользовательское пространство. Например, программа может начать передачу данных с диска в буфер внутри пользовательского процесса. Вызовет ли это какие-либо проблемы, если для реализации виртуальной памяти используется уплотнение? Аргументируйте.
10. Операционные системы, в которых допускаются проецируемые в память файлы, всегда требуют, чтобы файлы были отображены в границах страниц. Например, если у нас есть страницы по 4 К, файл может быть отображен, начиная с виртуального адреса 4096, но не с виртуального адреса 5000. Зачем это нужно?
11. При загрузке сегментного регистра в Pentium II вызывается соответствующий дескриптор, который загружается в невидимую часть сегментного регистра. Как вы думаете, почему разработчики Intel решили это сделать?
12. Программа в компьютере Pentium II обращается к локальному сегменту 10 со смещением 8000. Поле BASE сегмента 10 в локальной таблице дескрипторов содержит число 10000. Какой элемент таблицы страниц использует Pentium II? Каков номер страницы? Каково смещение?
13. Рассмотрите возможные алгоритмы для удаления сегментов в сегментированной памяти без страничной организации.
14. Сравните внутреннюю фрагментацию с внешней фрагментацией. Что можно сделать, чтобы улучшить каждую из них?
15. Супермаркеты часто сталкиваются с проблемой, сходной с замещением страниц в системах с виртуальной памятью. В супермаркетах есть фиксированная площадь пространства на полках, куда требуется помещать все больше и больше различных товаров. Если поступил новый важный продукт, например питание для собак очень высокого качества, какой-либо другой продукт нужно убрать, чтобы освободить место для нового продукта. Мы знаем два алгоритма: LRU и FIFO. Какой из них вы бы предпочли?
16. Почему блоки кэш-памяти всегда намного меньше, чем страницы в виртуальной памяти (бывает даже, что в 100 раз меньше)?
17. Почему многие системы файлов требуют, чтобы файл перед прочтением явным образом открывался с помощью системного вызова *open*?
18. Сравните применение битового отображения и списка неиспользованных пространств для слежения за свободным пространством на диске. Диск состоит из 800 цилиндров, на каждом из которых расположено 5 дорожек по 32 сектора. Сколько понадобится «дырок», чтобы список «дырок» (список свободной памяти) стал больше, чем битовое отображение? Предполагается, что единичный блок — это сектор и что для «дырки» требуется 32-битный элемент таблицы.
19. Чтобы сделать некоторые прогнозы относительно производительности диска, нужно иметь модель распределения памяти. Предположим, что диск рассматривается как линейное адресное пространство из N>>1 секторов. Здесь сначала идет последовательность блоков данных, затем неиспользованное пространство, затем другая последовательность блоков данных и т. д. Эмпирические измерения показывают, что вероятностные распределения для

длин данных и неиспользованных пространств одинаковы, причем для каждого из них вероятность быть i секторов составляет 2^{-i}. Каково при этом ожидаемое число «дырок» на диске?

20. На определенной машине программа может создавать столько файлов, сколько ей нужно, и все файлы могут увеличиваться в размерах во время выполнения программы, причем операционная система не получает никаких дополнительных данных об их конечном размере. Как вы думаете, хранятся ли файлы в последовательных секторах? Поясните.

21. Рассмотрим один метод реализации команд для работы с семафорами. Всякий раз, когда центральный процессор собирается совершить команду up или down над семафором (семафор — это целочисленная переменная в памяти), сначала он устанавливает приоритет центрального процессора таким образом, чтобы блокировать все прерывания. Затем он вызывает из памяти семафор, изменяет его и в соответствии с этим совершает переход. После этого он снова снимает запрет с прерываний. Будет ли этот метод работать, если:

    а. Существует один центральный процессор, который переключается между процессами каждые 100 миллисекунд?

    б. Два центральных процессора разделяют общую память, в которой расположен семафор?

22. Компания, разрабатывающая операционные системы, получает жалобы от своих клиентов по поводу последней разработки, которая поддерживает операции с семафорами. Клиенты решили, что аморально со стороны процессов приостанавливать свою работу (то есть спать на работе). Чтобы угодить своим клиентам, компания решила добавить третью операцию, *peek*. Эта операция просто проверяет семафор, но не изменяет его и не блокирует процесс. Таким образом, программы сначала проверяют, можно ли делать над семафором операцию *down*. Будет ли эта идея работать, если семафор используют три и более процессов? А если два процесса?

23. Составьте таблицу, в которой в виде функции от времени от 0 до 1000 миллисекунд показано, какие из трех процессов P1, P2 и P3 работают, а какие блокированы. Все три процесса выполняют команды up и down над одним и тем же семафором. Если два процесса блокированы и совершается команда up, то запускается процесс с меньшим номером, то есть P1 имеет преимущество над P2 и P3 и т. д. Изначально все три процесса работают, а значение семафора равно 1.

    При t=100 P1 совершает down.

    При t=200 P1 совершает down.

    При t=300 P1 совершает up.

    При t=400 P1 совершает down.

    При t=500 P1 совершает down.

    При t=600 P1 совершает up.

    При t=700 P1 совершает down.

    При t=800 P1 совершает up.

    При t=900 P1 совершает up.

24. В системе бронирования билетов на авиарейсы необходимо быть уверенным в том, что пока один процесс использует файл, никакой другой процесс не может использовать этот же файл. В противном случае два разных процесса, которые работают на два разных агентства по продаже билетов, могут продать последнее оставшееся место двум пассажирам. Разработайте метод синхронизации с использованием семафоров, чтобы точно знать, что только один процесс в конкретный момент времени может получать доступ к файлу (предполагается, что процессы подчиняются правилам).
25. Чтобы сделать возможной реализацию семафоров на компьютере с несколькими процессорами, которые разделяют общую память, разработчики включают в машину команду для проверки и блокирования. Команда TSL X проверяет ячейку X. Если содержание равно 0, семафоры устанавливаются на 1 за один неделимый цикл памяти, а следующая команда пропускается. Если содержание ячейки не равно 0, TSL работает как пустая операция. Используя TSL, можно написать процедуры *lock* и *unlock* со следующими свойствами: *lock*(x) проверяет, заперт ли x. Если нет, эта процедура запирает x и возвращает управление; *unlock* отменяет существующую блокировку. Если x уже заперт, процедура просто ждет, пока он не освободится, и только после этого запирает x и возвращает управление. Если все процессы запирают таблицу семафоров перед ее использованием, то в определенный момент времени только один процесс может производить операции с переменными и указателями, что предотвращает состояние гонок. Напишите процедуры *lock* и *unlock* на ассемблере.
26. Каково будет значение *in* и *out* для кольцевого буфера длиной в 65 слов после каждой из следующих операций? Изначально значения *in* и *out* равны 0.
    а. 22 слова помещаются в буфер;
    б. 9 слов удаляются из буфера;
    в. 40 слов помещаются в буфер;
    г. 17 слов удаляются из буфера;
    д. 12 слов помещаются в буфер;
    е. 45 слов удаляются из буфера;
    ж. 8 слов помещаются в буфер;
    з. 11 слов удаляются из буфера.
27. Предположим, что одна из версий UNIX использует 2 К блоков на диске и хранит 512 адресов диска на каждый блок косвенной адресации (обычной косвенной адресации, двойной и тройной). Каков будет максимальный размер файла? Предполагается, что размер указателей файла составляет 64 бита.
28. Предположим, что системный вызов UNIX

    ```
    unlink("/usr/ast/bin/game3")
    ```

    был выполнен в контексте рис. 6.27. Опишите подробно, какие изменения произойдут в системе директорий.
29. Представьте, что вам нужно реализовать систему UNIX на микрокомпьютере, где основной памяти недостаточно. После долгой работы она все еще не вполне влезает в память, и вы выбираете системный вызов наугад, чтобы

пожертвовать им для общего блага. Вы выбрали системный вызов *pipe*, который создает каналы для передачи потоков байтов от одного процесса к другому. Возможно ли после этого как-то изменить ввод-вывод? Что вы можете сказать о конвейерах? Рассмотрите проблемы и возможные решения.

30. Комиссия по защите дескрипторов файлов выдвинула протест против системы UNIX, потому что когда эта система возвращает дескриптор файла, она всегда возвращает самый маленький номер, который в данный момент не используется. Следовательно, едва ли когда-нибудь будут использоваться дескрипторы файлов с большими номерами. Комиссия настаивает на том, чтобы система возвращала дескриптор с самым маленьким номером из тех, которые еще не использовались программой, а не из тех, которые не используются в данный момент. Комиссия утверждает, что эту идею легко реализовать, это не повлияет на существующие программы и, кроме того, это будет гораздо справедливее по отношению к дескрипторам. А что вы думаете по этому поводу?
31. В системе NT можно составить список управления доступом таким образом, чтобы один пользователь не имел доступа ни к одному из файлов, а все остальные имели полный доступ к ним. Как это можно реализовать?
32. Опишите два способа программирования работы процессора-производителя и процессора-потребителя с использованием общих буферов и семафоров в NT. Подумайте о том, как можно реализовать разделенный буфер в каждом из двух случаев.
33. Работу алгоритмов замещения страниц обычно проверяют путем моделирования. Предположим, что вам нужно написать моделирующую программу для виртуальной памяти со страничной организацией для машины, содержащей 64 страницы по 1 Кбайт. Программа должна поддерживать одну таблицу из 64 элементов, один элемент на страницу. Каждый элемент таблицы содержит номер физической страницы, который соответствует данной виртуальной странице. Моделирующая программа должна считывать файл, содержащий виртуальные адреса в десятичной системе счисления, по одному адресу на строку. Если соответствующая страница находится в памяти, просто записывайте наличие страницы. Если ее нет в памяти, вызовите процедуру замещения страниц, чтобы выбрать страницу, которую можно выкинуть (то есть элемент таблицы, который нужно переписать), и записывайте отсутствие страницы. Никакой передачи страниц не происходит. Создайте файл, состоящий из непоследовательных адресов, и проверьте производительность работы двух алгоритмов: LRU и FIFO. А теперь создайте файл адресов, в котором x процентов адресов находятся на 4 байта выше, чем предыдущие. Проведите тесты для различных значений x и сообщите о полученных результатах.
34. Напишите программу для UNIX или NT, которая на входе получает имя директории. Программа должна печатать список файлов этой директории, каждый файл на отдельной строке, а после имени файла должен печататься размер файла. Имена файлов должны располагаться в том порядке, в котором они появляются в директории. Неиспользованные слоты в директории должны выводиться с пометой (неиспользованный).

Глава 7
Уровень языка ассемблера

В четвертой, пятой и шестой главах мы обсуждали три уровня, которые имеются в большинстве современных компьютеров. В этой главе речь пойдет о еще одном уровне, который также присутствует практически во всех современных машинах. Это уровень языка ассемблера. Уровень языка ассемблера существенно отличается от трех предыдущих, поскольку он реализуется с помощью трансляции, а не с помощью интерпретации.

Программы, которые преобразуют пользовательские программы, написанные на каком-либо определенном языке, в другой язык, называются **трансляторами**. Язык, на котором изначально написана программа, называется **входным языком**, а язык, на который транслируется эта программа, называется **выходным языком**. Входной язык и выходной язык определяют уровни. Если имеется процессор, который может выполнять программы, написанные на входном языке, то нет необходимости транслировать исходную программу на другой язык.

Трансляция используется в том случае, если есть аппаратный или программный процессор для выходного языка и нет процессора для входного языка. Если трансляция выполнена правильно, то оттранслированная программа будет давать точно такие же результаты, что и исходная программа (если бы существовал подходящий для нее процессор). Следовательно, можно организовать новый уровень, который сначала будет транслировать программы на выходной уровень, а затем выполнять полученные программы.

Важно понимать разницу между трансляцией и интерпретацией[1]. При трансляции исходная программа на входном языке не выполняется сразу. Сначала она преобразуется в эквивалентную программу, так называемую **объектную программу**, или **исполняемую двоичную программу**, которая выполняется только после завершения трансляции. При трансляции нужно пройти следующие два шага:

1. Создание эквивалентной программы на выходном языке.
2. Выполнение полученной программы.

Эти два шага выполняются не одновременно. Второй шаг начинается только после завершения первого. В интерпретации есть только один шаг: выполнение исходной программы. Никакой эквивалентной программы порождать не нужно, хотя иногда исходная программа преобразуется в промежуточную форму (например, в код Java) для упрощения интерпретации.

---

[1] В отечественной литературе принято и интерпретацию, и компиляцию (именно компиляцию автор здесь называет трансляцией) называть трансляцией. Другими словами, трансляторы могут быть либо компиляторами, либо интерпретаторами. — *Примеч. научн. ред.*

Во время выполнения объектной программы задействовано только три уровня: микроархитектурный уровень, уровень команд и уровень операционной системы. Следовательно, во время работы программы в памяти компьютера можно найти три программы: пользовательскую объектную программу, операционную систему и микропрограмму (если она есть). Никаких следов исходной программы не остается. Таким образом, число уровней, присутствующих при выполнении программы, может отличаться от числа уровней, присутствующих до трансляции. Следует отметить, что хотя мы определяем уровень по командам и языковым конструкциям, доступным программистам этого уровня (а не по технологии реализации), некоторые авторы иногда проводят различие между уровнями, реализованными интерпретаторами, и уровнями, реализованными при трансляции.

Введение в язык ассемблера

Трансляторы можно разделить на две группы в зависимости от отношения между входным и выходным языком. Если входной язык является символической репрезентацией числового машинного языка, то транслятор называется **ассемблером**, а входной язык называется **языком ассемблера**. Если входной язык является языком высокого уровня (например, Java или C), а выходной язык является либо числовым машинным языком, либо символической репрезентацией последнего, то транслятор называется **компилятором**.

Что такое язык ассемблера?

Язык ассемблера — это язык, в котором каждое высказывание соответствует ровно одной машинной команде. Иными словами, существует взаимно однозначное соответствие между машинными командами и операторами в программе на языке ассемблера. Если каждая строка в программе на языке ассемблера содержит ровно один оператор и каждое машинное слово содержит ровно одну команду, то программа на языке ассемблера в n строк произведет программу на машинном языке из n слов.

Мы используем язык ассемблера, а не программируем на машинном языке (в шестнадцатеричной системе счисления), поскольку на языке ассемблера программировать гораздо проще. Использовать символьные имена и адреса вместо двоичных и восьмеричных намного удобнее. Многие могут запомнить, что обозначениями для сложения (add), вычитания (subtract), умножения (multiply) и деления (divide) служат команды `ADD`, `SUB`, `MUL` и `DIV`, но мало кто может запомнить соответствующие числа, которые использует машина. Программисту на языке ассемблера нужно знать только символические названия, поскольку ассемблер транслирует их в машинные команды.

Это утверждение касается и адресов. Программист на языке ассемблера может дать имена ячейкам памяти, и уже ассемблер должен будет выдавать правильные числа. Программист на машинном языке всегда должен работать с числовыми номерами адресов. Сейчас уже нет программистов, которые пишут программы на машинном языке, хотя несколько десятилетий назад до изобретения ассемблеров программы именно так и писались.

Язык ассемблера имеет несколько особенностей, отличающих его от языков высокого уровня. Во-первых, это взаимно однозначное соответствие между высказываниями языка ассемблера и машинными командами (об этом мы уже говорили). Во-вторых, программист на языке ассемблера имеет доступ ко всем объектам и командам, присутствующим на целевой машине. У программистов на языках высокого уровня такого доступа нет. Например, если целевая машина содержит бит переполнения, программа на языке ассемблера может проверить его, а программа на языке Java не может. Программа на языке ассемблера может выполнить любую команду из набора команд целевой машины, а программа на языке высокого уровня не может. Короче говоря, все, что можно сделать в машинном языке, можно сделать и на языке ассемблера, но многие команды, регистры и другие объекты недоступны для программиста, пишущего программы на языке высокого уровня. Языки для системного программирования (например C) часто занимают промежуточное положение. Они обладают синтаксисом языка высокого уровня, но при этом с точки зрения возможностей доступа ближе к языку ассемблера.

Наконец, программа на языке ассемблера может работать только на компьютерах одного семейства, а программа, написанная на языке высокого уровня, потенциально может работать на разных машинах. Возможность переносить программное обеспечение с одной машины на другую очень важна для многих прикладных программ.

Зачем нужен язык ассемблера?

Язык ассемблера довольно труден. Написание программы на языке ассемблера занимает гораздо больше времени, чем написание той же программы на языке высокого уровня. Кроме того, очень много времени занимает отладка.

Но зачем же тогда вообще писать программы на языке ассемблера? Есть две причины: производительность и доступ к машине. Во-первых, профессиональный программист языка ассемблера может составить гораздо меньшую по размеру программу, которая будет работать гораздо быстрее, чем программа, написанная на языке высокого уровня. Для некоторых программ скорость и размер весьма важны. Многие встроенные прикладные программы, например программы в кредитных карточках, сотовых телефонах, драйверах устройств, а также процедуры BIOS попадают в эту категорию.

Во-вторых, некоторым процедурам требуется полный доступ к аппаратному обеспечению, что обычно невозможно сделать на языке высокого уровня. В эту категорию попадают прерывания и обработчики прерываний в операционных системах, а также контроллеры устройств во встроенных системах, работающих в режиме реального времени.

Первая причина (достижение высокой производительности) является более важной, поэтому мы рассмотрим ее подробнее. В большинстве программ лишь небольшой процент всего кода отвечает за большой процент времени выполнения программы. Обычно 1% программы отвечает за 50% времени выполнения, а 10% программы отвечает за 90% времени выполнения.

Предположим, что для написания программы на языке высокого уровня требуется 10 человеко-лет и что полученной программе требуется 100 секунд, чтобы

выполнить некоторую типичную контрольную задачу. (**Контрольная задача** — это программа проверки, которая используется для сравнения компьютеров, компиляторов и т. п.). Написание всей программы на языке ассемблера может занять 50 человеко-лет. Полученная в результате программа будет выполнять контрольную задачу примерно за 33 секунды, поскольку хороший программист может оказаться в три раза умнее компилятора (хотя об этом можно спорить бесконечно). Ситуация проиллюстрирована в табл. 7.1.

Так как только крошечная часть программы отвечает за большую часть времени выполнения этой программы, возможен другой подход. Сначала программа пишется на языке высокого уровня. Затем проводится ряд измерений, чтобы определить, какие части программы отвечают за большую часть времени выполнения. Для таких измерений обычно используется системный тактовый генератор. С его помощью можно узнать, сколько времени затрачивается на каждую процедуру, сколько раз выполняется каждый цикл и т. п.

Предположим, что 10% программы отвечает за 90% времени ее выполнения. Это значит, что из всех 100 секунд работы 90 секунд проводится в этих 10%, а 10 секунд — в оставшихся 90% программы. Эти 10% программы можно усовершенствовать, переписав их на язык ассемблера. Этот процесс называется **настройкой** (tuning). Он проиллюстрирован в табл. 7.1. На переделку основных процедур потребуется еще 5 лет, но время выполнения программы сократится с 90 секунд до 30 секунд.

Таблица 7.1. Сравнение программирования на языке ассемблера и на языке высокого уровня (с настройкой и без настройки)

	Количество человеко-лет, затрачиваемых на написание программы	**Время выполнения программы в секундах**
Язык ассемблера	50	33
Язык высокого уровня	10	100
Смешанный подход до настройки		
Критические 10%	1	90
Остальные 90%	9	10
Всего	10	100
Смешанный подход после настройки		
Критические 10%	6	30
Остальные 90%	9	10
Всего	15	40

Сравним этот смешанный подход, в котором используется и язык ассемблера, и язык высокого уровня, с подходом, в котором применяется только язык ассемблера (табл. 7.1). При втором подходе программа работает примерно на 20% быстрее (33 секунды против 40 секунд), но более чем за тройную цену (50 человеко-лет против 15). Более того, у смешанного подхода есть еще одно преимущество: гораздо проще переделать в код ассемблера уже отлаженную процедуру, написанную на языке высокого уровня, чем писать процедуру на языке ассемблера с нуля.

Отметим, что если бы написание программы занимало только 1 год, соотношение между смешанным подходом и подходом, при котором используется только язык ассемблера, составляло бы 4:1 в пользу смешанного подхода.

Программист, который использует язык высокого уровня, не занят перемещением битов и может так решить задачу, так построить программу, что в конце концов достигнет действительно большого увеличения производительности. А программисты, пишущие программы на языке ассемблера, обычно стараются так построить команды, чтобы сэкономить несколько циклов, поэтому у них такой ситуации возникнуть не может.

Расскажем о двух экспериментах, проведенных во время разработки системы MULTICS. Грехем [49] описал процедуру PL/I, за три месяца переделанную в новую версию, которая была в 26 раз меньше и работала в 50 раз быстрее, чем исходная. Он описал еще одну процедуру PL/L, которая получилась в 20 раз меньше исходной и работала в 40 раз быстрее, чем исходная, после двух месяцев работы. Корбато [27] описал процедуру PL/I, размер кода которой был сокращен с 50 000 слов до 1000 слов менее чем за месяц, а контроллер уменьшен с 65 000 до 30 000 слов с увеличением производительности в 8 раз за 4 месяца. Здесь важно понимать, что у программистов языков высокого уровня глобальный подход к тому, что они делают, поэтому они гораздо быстрее могут разработать лучший алгоритм.

Однако, несмотря на все это, существует по крайней мере 4 веские причины для изучения языка ассемблера. Во-первых, желательно уметь писать программы на языке ассемблера, поскольку успех или неудача большого проекта может зависеть от того, можно ли повысить производительность какой-то важной процедуры в 2 или 3 раза.

Во-вторых, язык ассемблера может быть единственным возможным выходом из-за недостатка памяти. Кредитные карты, например, содержат центральный процессор, но у них нет мегабайта памяти и жесткого диска. Однако они должны выполнять сложные вычисления при наличии ограниченных ресурсов. Процессоры, встроенные в электроприборы, часто имеют минимальное количество памяти, поскольку они должны быть достаточно дешевыми. Различные электронные устройства, работающие на батарейках, обычно содержат очень маленькую память, поэтому здесь тоже нужен эффективный код.

В-третьих, компилятор должен либо давать на выходе программу, которая используется ассемблером, либо самостоятельно выполнять процесс ассемблирования. Таким образом, понимание языка ассемблера существенно для понимания того, как работает компьютер. И вообще, кто-то ведь должен писать компилятор и его ассемблер.

Наконец, изучение языка ассемблера дает прекрасное представление о реальной машине. Для тех, кто изучает архитектуру компьютеров, написание программы на языке ассемблера — единственный способ узнать, что собой представляет машина на архитектурном уровне.

Формат оператора в языке ассемблера

Хотя структура оператора в языке ассемблера отражает структуру соответствующей машинной команды, языки ассемблера для разных машин и разных уровней во многом сходны друг с другом, что позволяет говорить о языке ассемблера вооб-

ще. В таблицах 7.2–7.4 показаны фрагменты программ на языке ассемблера для Pentium II, Motorola 680x0 и (Ultra)SPARC. Все эти программы выполняют вычисление N=I+J. Во всех трех примерах операторы над пропуском в таблице выполняют вычисление. Операторы под пропуском — это указания ассемблеру зарезервировать память для переменных I, J и N. Последние не являются символьными репрезентациями машинных команд.

Таблица 7.2. Вычисление выражения N=I+J в Pentium II

Метка	Код операции	Операнды	Комментарии
FORMULA:	MOV	EAX,I	; регистр EAX=I
	ADD	EAX,J	; регистр EAX=I+J
	MOV	N,EAX	; N=I+J
I	DW	3	; резервируем 4 байта и устанавливаем значение 3
J	DW	4	; резервируем 4 байта и устанавливаем значение 4
N	DW	0	; резервируем 4 байта и устанавливаем значение 0

Таблица 7.3. Вычисление выражения N=I+J в Motorola 680x0

Метка	Код операции	Операнды	Комментарии
FORMULA:	MOVE.L	I,D0	; регистр D0=I
	ADD.L	J,D0	; регистр D0=I+J
	MOVE.L	D0,N	; N=I+J
I	DC.L	3	; резервируем 4 байта и устанавливаем значение 3
J	DC.L	4	; резервируем 4 байта и устанавливаем значение 4
N	DC.L	0	; резервируем 4 байта и устанавливаем значение 0

Таблица 7.4. Вычисление выражения N=I+J в SPARC

Метка	Код операции	Операнды	Комментарии
FORMULA:	SETHI	%HI(I),%R1	! R1 = старшие биты адреса I
	LD	[%R1+%LO(I)],%R1	! R1=I
	SETHI	%HI(J),%R2	! R2 = старшие биты адреса J
	LD	[%R2+%LO(J)],%R2	! R2=J
	NOP		! ждем прибытия J из памяти
	ADD	%R1,%R2,%R2	! R2=R1+R2
	SETHI	%HI(N),%R1	! R1 = старшие биты адреса N
	ST	%R2, [%R1+%LO(N)]	
I:	.WORD 3	3	! резервируем 4 байта и устанавливаем знач. 3
J:	.WORD 4	4	! резервируем 4 байта и устанавливаем знач. 4
N:	.WORD 0	0	! резервируем 4 байта и устанавливаем знач. 0

Для компьютеров семейства Intel существует несколько ассемблеров, которые отличаются друг от друга по синтаксису. В этой книге мы будем использовать язык ассемблера Microsoft MASM. Мы будем говорить о процессоре Pentium II, но все, что мы будем обсуждать, применимо и к процессорам 386, 486, Pentium и Pentium Pro. Для процессора SPARC мы будем использовать ассемблер Sun. Все это также применимо к более ранним 32-битным версиям. В книге коды операций и регистры всегда обозначаются прописными буквами, причем не только в ассемблере для Pentium II, как это обычно принято, но и в ассемблере Sun, где по соглашению используются строчные буквы.

Высказывания языка ассемблера состоят из четырех полей: поля метки, поля операции, поля операндов и поля комментариев. Метки используются для того, чтобы обеспечить символические имена для адресов памяти. Они нужны для того, чтобы можно было совершить переход к командам. Они также нужны для слов с данными, чтобы по символическому имени можно было получить доступ к тому месту, где они хранятся. Если высказывание снабжено меткой, то эта метка обычно располагается в колонке 1.

В каждом из трех примеров есть 4 метки: FORMULA, I, J и N. Отметим, что в языках ассемблера для SPARC после каждой метки нужно ставить двоеточие, а для Motorola — нет. В компьютерах Intel двоеточия ставятся только после меток команд, но не после меток данных. Данное различие вовсе не является фундаментальным. Разработчики разных ассемблеров имеют разные вкусы. Архитектура машины никак не определяет тот или иной выбор. Единственное преимущество двоеточия состоит в том, что метку можно писать на отдельной строке, а код операции — на следующей строке в колонке 1. Это упрощает работу компилятора: без двоеточия нельзя было бы отличить метку на отдельной строке от кода операции на отдельной строке.

В некоторых ассемблерах длина метки ограничена до 6 или 8 символов. А в большинстве языков высокого уровня длина имен произвольна. Длинные и хорошо подобранные имена упрощают чтение и понимание программы другими людьми.

В каждой машине содержится несколько регистров, но всем им даны совершенно разные названия. Регистры в Pentium II называются EAX, EBX, ECX и т. д. Регистры в Motorola называются D0, D1, D2. Регистры в машине SPARC имеют несколько названий. Здесь для их обозначения мы будем использовать %R1 и %R2.

В поле кода операции содержится либо символическая аббревиатура этого кода (если высказывание является символической репрезентацией машинной команды), либо команда для самого ассемблера. Выбор имени — дело вкуса, и поэтому разные разработчики языков ассемблера называют их по-разному. Разработчики ассемблера Intel решили использовать обозначение `MOV` и для загрузки регистра из памяти, и для сохранения регистра в память. Разработчики ассемблера Motorola выбрали обозначение `MOVE` для обеих операций. А разработчики ассемблера SPARC решили использовать `LD` для первой операции и `ST` для второй. Очевидно, что выбор названий в данном случае никак не связан с архитектурой машины.

Напротив, необходимость использовать две машинные команды для доступа к памяти объясняется устройством архитектуры SPARC, поскольку виртуальные адреса могут быть 32-битными (как в SPARC Version 8) и 44-битными (как в SPARC

Version 9), а команды могут содержать максимум 22 бита данных. Следовательно, чтобы передать все биты полного виртуального адреса, всегда требуется две команды. Команда

```
SETHI %HI(I),%R1
```

обнуляет старшие 32 бита и младшие 10 битов 64-битного регистра R1, а затем помещает старшие 22 бита 32-битного адреса переменной I в регистр R1 в битовые позиции с 10 по 31. Следующая команда

```
LD[%R1+%LO(I)],%R1
```

складывает R1 и младшие 10 битов адреса I (в результате чего получается полный адрес I), вызывает данное слово из памяти и помещает его в регистр R1.

Процессоры семейства Pentium, 680x0 и SPARC — все допускают операнды разной длины (типа byte (байт), word (слово) и long). Каким образом ассемблер определит, какую длину использовать? И опять разработчики ассемблера приняли разные решения. В Pentium II регистры разной длины имеют разные названия. Так, для перемещения 32-битных элементов используется название EAX, для 16-битных — AX, а для 8-битных — AL и AH. Разработчики ассемблера Motorola решили прибавлять к каждому коду операции суффикс .L для типа long, .W — для типа word и .B для типа byte. В SPARC для операндов разной длины используются разные коды операций (например, для загрузки байта, полуслова (halfword) и слова в 64-битный регистр используются коды операций `LDSB`, `LDSH` и `LDSW` соответственно). Как видите, разработка языка произвольна.

Три ассемблера, которые мы рассматриваем, различаются по способу резервирования пространства для данных. Разработчики языка ассемблера для Intel выбрали `DW` (Define Word — определить слово). Позднее был введен альтернативный вариант `.WORD`. В Motorola используется `DC` (Define Constant — определить константу). Разработчики SPARC с самого начала предпочли `.WORD`. И слова различия произвольны.

В поле операндов определяются адреса и регистры, которые являются операндами для машинной команды. В поле операндов команды целочисленного сложения сообщается, что и к чему нужно прибавить. Поле операндов команд перехода определяет, куда нужно совершить переход. Операндами могут быть регистры, константы, ячейки памяти и т. д.

В поле комментариев приводятся пояснения о действиях программы. Они могут понадобиться программистам, которые будут использовать и переделывать чужую программу, или программисту, который изначально писал программу и возвратился к работе над ней через год. Программа на ассемблере без таких комментариев совершенно непонятна программистам (даже автору этой программы). Комментарии нужны только человеку. Они никак не влияют на работу программы.

Директивы

Программа на языке ассемблера должна не только определять, какие машинные команды нужно выполнить, но и содержать команды, которые должен выполнять сам ассемблер (например, потребовать от него определить местонахождение какой-либо сохраненной информации или выдать новую страницу листинга). Команды для ассемблера называются **псевдокомандами** или **директивами ассем-**

блера. Мы уже видели одну типичную псевдокоманду DW (см. табл. 7.2). В табл. 7.5 приведены некоторые другие псевдокоманды (директивы). Они взяты из ассемблера MASM для семейства Intel.

Таблица 7.5. Некоторые директивы ассемблера MASM

Директива	Значение
SEGMENT	Начинает новый сегмент (текста, данных и т.п.) с определенными атрибутами
ENDS	Завершает текущий сегмент
ALIGN	Контролирует выравнивание следующей команды или данных
EQU	Определяет новый символ, равный данному выражению
DB	Выделяет память для одного или нескольких байтов
DD	Выделяет память для одного или нескольких 16-битных полуслов
DW	Выделяет память для одного или нескольких 32-битных слов
DQ	Выделяет память для одного или нескольких 64-битных двойных слов
PROC	Начинает процедуру
ENDP	Завершает процедуру
MACRO	Начинает макроопределение
ENDM	Завершает макроопределение
PUBLIC	Экспортирует имя, определенное в данном модуле
EXTERN	Импортирует имя из другого модуля
INCLUDE	Вызывает другой файл и включает его в текущий файл
IF	Начинает условную компоновку программы на основе данного выражения
ELSE	Начинает условную компоновку программы, если условие IF над директивой не выполнено
ENDIF	Завершает условную компоновку программы
COMMENT	Определяет новый отделитель комментариев
PAGE	Совершает принудительный обрыв страницы в листинге
END	Завершает программу ассемблирования

Директива SEGMENT начинает новый сегмент, а директива ENDS завершает его. Разрешается начинать текстовый сегмент, затем начинать сегмент данных, затем переходить обратно к текстовому сегменту и т. д.

Директива ALIGN переводит следующую строку (обычно данные) в адрес, который делим на аргумент данной директивы. Например, если текущий сегмент уже содержит 61 байт данных, тогда следующим адресом после ALIGN 4 будет адрес 64.

Директива EQU дает символическое название некоторому выражению. Например, после записи

```
BASE EQU 1000
```

символ BASE можно использовать вместо 1000. Выражение, которое следует за EQU, может содержать несколько символов, соединенных арифметическими и другими операторами, например:

```
LIMIT EQU 4 * BASE + 2000
```

Большинство ассемблеров, в том числе MASM, требуют, чтобы символ был определен в программе до появления в некотором выражении.

Следующие 4 директивы DB, DD, DW и DQ распределяют память для одной или нескольких переменных размером 1, 2, 4 и 8 байтов соответственно. Например,

```
TABLE DB 11, 23, 49
```

выделяет пространство для 3 байтов и присваивает им начальные значения 11, 23 и 49 соответственно. Эта директива, кроме того, определяет символ TABLE, равный тому адресу, где хранится число 11.

Директивы PROC и ENDP определяют начало и конец процедур языка ассемблера. Процедуры в языке ассемблера выполняют ту же функцию, что и в языках программирования высокого уровня. Директивы MACRO и ENDM определяют начало и конец макроса. О макросах мы будем говорить ниже.

Далее идут директивы PUBLIC и EXTERN. Программы часто пишут в виде совокупности файлов. Часто процедуре, находящейся в одном файле, нужно вызвать процедуру или получить доступ к данным, определенным в другом файле. Чтобы такие отсылки между файлами стали возможными, обозначение (имя), которое нужно сделать доступным для других файлов, экспортируется с помощью директивы PUBLIC. Чтобы ассемблер не ругался по поводу использования символа, который не определен в данном файле, этот символ может быть объявлен внешним (EXTERN), это сообщит ассемблеру, что символ определен в каком-то другом файле. Символы, которые не определены ни в одной из этих директив, используются только в пределах одного файла. Поэтому даже если символ FOO используется в нескольких файлах, это не вызовет никакого конфликта, поскольку этот символ локален по отношению к каждому файлу.

Директива INCLUDE приказывает ассемблеру вызвать другой файл и включить его в текущий файл. Такие включенные файлы часто содержат определения, макросы и другие элементы, необходимые для разных файлов.

Многие языки ассемблера, в том числе MASM, поддерживают условную компоновку программы. Например, программа

```
WORDSIZE EQU 16
IF WORDSIZE GT 16
WSIZE:  DW32
ELSE
WSIZE:  DW 16
ENDIF
```

выделяет в памяти одно 32-битное слово и вызывает его адрес WSIZE. Этому слову придается одно из значений: либо 32, либо 16 в зависимости от значения WORDSIZE (в данном случае 16). Такая конструкция может использоваться в программах для 16-битных машин (как 8088) или для 32-битных машин (как Pentium II). Если в начале и в конце машинозависимого кода поставить IF и ENDIF, а затем изменить одно определение, WORDSIZE, программу можно автоматически установить на один из двух размеров. Применяя такой подход, можно сохранять одну такую исходную программу для нескольких разных машин. В большинстве случаев все машинозависимые определения, такие как WORDSIZE, сохраняются в одном файле, причем для разных машин должны быть разные файлы. Путем включения файла с нужными определениями программу можно легко перекомпилировать на разные машины.

Директива COMMENT позволяет пользователю изменять символ комментария на что-либо отличное от точки с запятой. Директива PAGE используется для управления листингом программы. Наконец, директива END отмечает конец программы.

В ассемблере MASM есть еще много директив. Другие ассемблеры для Pentium II содержат другой набор директив, поскольку они определяются не в соответствии с архитектурой машины, а по желанию разработчиков ассемблера.

Макросы

Программистам на языке ассемблера часто приходится повторять одни и те же цепочки команд по несколько раз. Проще всего писать нужные команды всякий раз, когда они требуются. Но если последовательность достаточно длинная или если ее нужно повторять очень много раз, то это становится утомительным.

Альтернативный подход — оформить эту последовательность в процедуру и вызывать ее в случае необходимости. У такой стратегии тоже есть свои недостатки, поскольку в этом случае каждый раз придется выполнять специальную команду вызова процедуры и команду возврата. Если последовательности команд короткие (например, всего две команды), но используются часто, то вызов процедуры может сильно снизить скорость работы программы. Макросы являются простым и эффективным решением этой проблемы.

Макроопределение, макровызов и макрорасширение

Макроопределение — это способ дать имя куску текста. После того как макрос был определен, программист может вместо куска программы писать имя макроса. В сущности, макрос — это обозначение куска текста. В листинге 7.1 приведена программа на языке ассемблера для Pentium II, которая дважды меняет местами содержимое переменных p и q. Эти последовательности команд можно определить как макросы (листинг 7.2). После определения макроса каждое имя SWAP в программе замещается следующими четырьмя строками:

```
MOV EAX,P
MOV EBX,Q
MOV Q,EAX
MOV P,EBX
```

Программист определил SWAP как обозначение для этих четырех операторов.

Хотя разные языки ассемблера используют немного разные записи для определения макросов, все они состоят из одних и тех же базовых частей:

1. Заголовок макроса, в котором дается имя определяемого макроса.
2. Текст, в котором приводится тело макроса.
3. Директива, которая завершает определение (например, ENDM).

Когда ассемблер наталкивается на макроопределение в программе, он сохраняет его в таблице макроопределений для последующего использования. Всякий раз, когда в программе в качестве кода операции появляется макрос (в нашем примере SWAP), ассемблер замещает его телом макроса. Использование имени макроса в качестве кода операции называется **макровызовом**, а его замещение телом макроса называется **макрорасширением**.

Листинг 7.1. Код на языке ассемблера, в котором переменные p и q дважды меняются местами (без использования макроса)

```
MOV EAX,P
MOV EBX,Q
MOV Q,EAX
MOV P,EBX

MOV EAX,P
MOV EBX,Q
MOV Q,EAX
MOV P,EBX
```

Листинг 7.2. Тот же код с использованием макроса

```
SWAP    MACRO
        MOV EAX,P
        MOV EBX,Q
        MOV Q,EAX
        MOV P,EBX
        ENDM

        SWAP

        SWAP
```

Макрорасширение происходит во время процесса ассемблирования, а не во время выполнения программы. Этот момент очень важен. Программы, приведенные в листингах 7.1 и 7.2, произведут один и тот же машинный код. По программе на машинном языке невозможно определить, использовались ли макросы при ее порождении. После завершения макрорасширения ассемблер отбрасывает макрорасширения. В полученной программе никаких признаков макросов не остается.

Макровызовы не следует путать с вызовами процедур. Основное различие состоит в том, что макровызов — это команда ассемблеру заменить имя макроса телом макроса. Вызов процедуры — это машинная команда, которая вставлена в объектную программу и которая позднее будет выполнена, чтобы вызвать процедуру. В табл. 7.6 сравниваются макровызовы и вызовы процедур.

Таблица 7.6. Сравнение макровызовов и вызовов процедур

	Макровызов	**Вызов процедуры**
Когда совершается вызов программы?	Во время ассемблирования	Во время выполнения
Вставляется ли тело макроса или процедуры в объектную программу каждый раз, когда совершается вызов?	Да	Нет
Команда вызова процедуры вставляется в объектную программу, а затем выполняется?	Нет	Да
Нужно ли после вызова использовать команду возврата?	Нет	Да
Сколько копий тела макровызова или процедуры появляется в объектной программе?	Одна на макровызов	1

Можно считать, что процесс ассемблирования осуществляется в два прохода. На первом проходе сохраняются все макроопределения, а макровызовы расширяются. На втором проходе обрабатывается полученный в результате текст. Иными словами, исходная программа считывается, а затем трансформируется в другую программу, из которой удалены все макроопределения и в которой каждый макровызов замещен телом макроса. Полученная программа без макросов затем поступает в ассемблер.

Важно иметь в виду, что программа представляет собой цепочку символов. Это могут быть буквы, цифры, пробелы, знаки пунктуации и «возврат каретки» (переход на новую строку). Макрорасширение состоит в замене определенных подцепочек из этой цепочки другими цепочками. Макросредства — это способ манипулирования цепочками символов безотносительно их значений.

Макросы с параметрами

Макросредства, описанные ранее, можно использовать для сокращения программ, в которых часто повторяется точно одна и та же последовательность команд. Однако очень часто программа содержит несколько похожих, но не идентичных последовательностей команд (листинг 7.3). Здесь первая последовательность меняет местами P и Q, а вторая последовательность меняет местами R и S.

Листинг 7.3. Почти идентичные последовательности команд без использования макроса

```
MOV EAX,P
MOV EBX,Q
MOV Q,EAX
MOV P,EBX

MOV EAX,R
MOV EBX,S
MOV S,EAX
MOV R,EBX
```

Листинг 7.4. Те же последовательности с использованием макроса

```
CHANGE    MACRO P1,P2
              MOV EAX,P1
              MOV EBX,P2
              MOV P2,EAX
              MOV P1,EBX
              ENDM

              CHANGE P,Q

              CHANGE R,S
```

Для работы с такими почти идентичными последовательностями предусмотрены макроопределения, которые обеспечивают **формальные параметры**, и макровызовы, которые обеспечивают **фактические параметры**. Когда макрос расширяется, каждый формальный параметр, который появляется в теле макроса, замещается соответствующим фактическим параметром. Фактические параметры помещаются в поле операндов макровызова. В листинге 7.4. представлена программа из листинга 7.3, в которую включен макрос с двумя параметрами. Символы P1

и P2 — это формальные параметры. Во время расширения макроса каждый символ P1 внутри тела макроса замещается первым фактическим параметром, а символ P2 замещается вторым фактическим параметром. В макровызове

```
CHANGE P,Q
```

P — это первый фактический параметр, а Q — это второй фактический параметр. Таким образом, программы в листингах 7.3 и 7.4 идентичны.

Расширенные возможности

Большинство макропроцессоров содержат целый ряд расширенных особенностей, которые упрощают работу программиста на языке ассемблера. В этом разделе мы рассмотрим несколько расширенных особенностей MASM. Во всех ассемблерах есть одна проблема: дублирование меток. Предположим, что макрос содержит команду условного перехода и метку, к которой совершается переход. Если макрос вызывается два и более раз, метка будет дублироваться, что вызовет ошибку. Поэтому программист должен приписывать каждому вызову в качестве параметра отдельную метку. Другое решение (оно применяется в MASM) — объявлять метку локальной (LOCAL), при этом ассемблер автоматически будет порождать другую метку при каждом расширении макроса. В некоторых ассемблерах номерные метки автоматически считаются локальными.

MASM и большинство других ассемблеров позволяют определять макросы внутри других макросов. Эта особенность очень полезна в сочетании с условной компоновкой программы. Обычно один и тот же макрос определятся в обеих частях оператора IF:

```
M1  MACRO
IF WORDSIZE GT 16 M2     MACRO
...
ENDM
ELSE
M2  MACRO
...
ENDM
ENDIF
ENDM
```

В любом случае макрос M2 будет определен, но определение зависит от того, на какой машине ассемблируется программа: на 16-битной или на 32-битной. Если M1 не вызывается, макрос M2 вообще не будет определен.

Наконец, одни макросы могут вызывать другие макросы, в том числе самих себя. Если макрос рекурсивный, то есть вызывает самого себя, он должен передавать самому себе параметр, который изменяется при каждом расширении, а также проверять этот параметр и завершать рекурсию, когда параметр достигает определенного значения. В противном случае получится бесконечный цикл.

Реализация макросредств в ассемблере

Для реализации макросов ассемблер должен уметь выполнять две функции: сохранять макроопределения и расширять макровызовы. Мы рассмотрим эти функции по очереди.

Ассемблер должен сохранять таблицу всех имен макросов, в которой каждое имя сопровождается указателем на определение этого макроса, чтобы его можно было получить в случае необходимости. В одних ассемблерах предусмотрена отдельная таблица для имен макросов, а другие содержат общую таблицу, в которой находятся не только имена макросов, но и все машинные команды и директивы.

Когда встречается макроопределение, создается новый элемент таблицы с именем макроса, числом параметров и указателем на другую таблицу — таблицу макроопределений, где будет храниться тело макроса. Список формальных параметров тоже создается в это время. Затем считывается тело макроса и сохраняется в таблице макроопределений. Формальные параметры, которые встречаются в теле цикла, указываются специальным символом.

Ниже приведен пример внутреннего представления макроса CHANGE. В качестве символа возврата каретки используется точка с запятой, а в качестве символа формального параметра — амперсант.

```
MOV EAX,&P1;MOV EBX,&P2;MOV &P2EAX;MOV &P1,EBX;
```

В таблице макроопределений тело макроса представляет собой просто цепочку символов.

Во время первого прохода ассемблирования отыскиваются коды операций, а макросы расширяются. Всякий раз, когда встречается макроопределение, оно сохраняется в таблице макросов. При вызове макроса ассемблер временно приостанавливает чтение входных данных из входного устройства и начинает считывать сохраненное тело макроса. Формальные параметры, извлеченные из тела макроса, замещаются фактическими параметрами, которые предоставляются вызовом. Амперсант перед параметрами позволяет ассемблеру узнавать их.

Процесс ассемблирования

В следующих разделах мы опишем, как работает ассемблер. И хотя на каждой машине есть свой определенный ассемблер, отличный от других, процесс ассемблирования по сути один и тот же.

Двухпроходной ассемблер

Поскольку программа на языке ассемблера состоит из ряда операторов, на первый взгляд может показаться, что ассемблер сначала должен читать оператор, затем транслировать его на машинный язык и, наконец, переносить полученный машинный язык в файл, а соответствующий кусок листинга — в другой файл. Этот процесс будет повторяться до тех пор, пока вся программа не будет оттранслирована. Но, к сожалению, такая стратегия не работает.

Рассмотрим ситуацию, где первый оператор — переход к L. Ассемблер не может ассемблировать это оператор, пока не будет знать адрес L. L может находиться где-нибудь в конце программы, и тогда ассемблер не сможет найти этот адрес, не прочитав всю программу. Эта проблема называется **проблемой опережающей**

ссылки, поскольку символ L используется еще до того, как он определен (то есть было сделано обращение к символу, определение которого появится позднее).

Опережающие ссылки можно разрешать двумя способами. Во-первых, ассемблер действительно может прочитать программу дважды. Каждое прочтение исходной программы называется **проходом**, а транслятор, который читает исходную программу дважды, называется **двухпроходным транслятором**. На первом проходе собираются и сохраняются в таблице все определения символов, в том числе метки. К тому времени как начнется второй проход, значения символов уже известны, поэтому никакой опережающей ссылки не будет, и каждый оператор можно читать и ассемблировать. При этом требуется дополнительный проход по исходной программе, но зато такая стратегия относительно проста.

При втором подходе программа на языке ассемблера читается один раз и преобразуется в промежуточную форму, и эта промежуточная форма сохраняется в таблице в памяти. Затем совершает второй проход, но уже не по исходной программе, а по таблице. Если физической памяти (или виртуальной памяти) достаточно для этого подхода, то будет сэкономлено время, затрачиваемое на процесс ввода-вывода. Если требуется вывести листинг, тогда нужно сохранить полностью исходное выражение, включая комментарии. Если листинг не нужен, то промежуточную форму можно сократить, оставив только голые команды.

Еще одна задача первого прохода — сохранить все макроопределения и расширить вызовы по мере их появления. Следовательно, в одном проходе происходит и определение символов, и расширение макросов.

Первый проход

Главная функция первого прохода — построить **таблицу символов**, в которой содержатся значения всех имен. Символом может быть либо метка, либо значение, которому с помощью директивы приписывается определенное имя:

```
BUFSIZE EQU 8192
```

Приписывая значение символьному имени в поле метки команды, ассемблер должен знать, какой адрес будет иметь эта команда во время выполнения программы. Для этого ассемблер во время процесса ассемблирования сохраняет **счетчик адреса команд (ILC — Instruction Location Counter)** (специальную переменную). Эта переменная устанавливается на 0 в начале первого прохода и увеличивается после каждой обработанной команды на длину этой команды (табл. 7.7.). Пример написан для Pentium II. Мы не будем давать примеры для SPARC и Motorola, поскольку различия между языками ассемблера не очень важны и одного примера будет достаточно. Кроме того, язык ассемблера для SPARC неудобочитаем.

При первом проходе в большинстве ассемблеров используется по крайней мере 3 таблицы: таблица символьных имен, таблица директив и таблица кодов операций. В случае необходимости используется еще литеральная таблица. Таблица символьных имен содержит один элемент для каждого имени, как показано в табл. 7.8. Символьные имена либо используются в качестве меток, либо явным образом определяются (например, с помощью EQU). В каждом элементе таблицы символьных

имен содержится само имя (или указатель на него), его численное значение и иногда некоторая дополнительная информация. Она может включать:

1. Длину поля данных, связанного с символом.
2. Биты перераспределения памяти (которые показывают, изменяется ли значение символа, если программа загружается не в том адресе, в котором предполагал ассемблер).
3. Сведения о том, можно ли получить доступ к символу извне процедуры.

Таблица 7.7. Счетчик адреса команд используется для слежения за адресами команд. В данном примере операторы до MARIA занимают 100 байтов

Метка	Код операции	Операнды	Комментарии	Длина	Счетчик адреса команд
MARIA:	MOV	EAX, I	EAX=I	5	100
	MOV	EBX, J	EBX=J	6	105
ROBERTA:	MOV	ECX, K	ECX=K	6	111
	IMUL	EAX, EAX	EAX=I*I	2	117
	IMUL	EBX, EBX	EBX=J*J	3	119
	IMUL	ECX, ECX	ECX=K*K	3	122
MARILYN:	ADD	EAX, EBX	EAX=I*I+J*J	2	125
	ADD	EAX, ECX	EAX=I*I+J*J+K*K	2	127
STEPHANY:	JMP	DONE	Переход к DONE	5	129

Таблица 7.8. Таблица символьных имен для программы из табл. 7.7.

Символьное имя	Значение	Прочая информация
MARIA	100	
ROBERTA	111	
MARILYN	125	
STEPHANY	129	

В таблице кодов операций предусмотрен по крайней мере один элемент для каждого символического кода операции в языке ассемблера (табл. 7.9). В каждом элементе таблицы содержится символический код операции, два операнда, числовое значение кода операции, длина команды и номер типа, по которому можно определить, к какой группе относится код операции (коды операций делятся на группы в зависимости от числа и типа операндов).

Таблица 7.9. Некоторые элементы таблицы кодов операций для ассемблера Pentium II

Код операции	Первый операнд	Второй операнд	Шестнадцатеричный код	Длина команды	Класс команды
AAA	—	—	37	1	6
ADD	EAX	immed32	05	5	4
ADD	reg	reg	01	2	19
AND	EAX	immed32	25	5	4
AND	reg	reg	21	2	19

В качестве примера рассмотрим код операции ADD. Если команда ADD в качестве первого операнда содержит регистр EAX, а в качестве второго — 32-битную константу (immed32), то используется код операции 0x05, а длина команды составляет 5 байтов. Если используется команда ADD с двумя регистрами в качестве операндов, то длина команды составляет 2 байта, а код операции будет равен 0x01. Все комбинации кодов операций и операндов, которые соответствуют данному правилу, будут отнесены к классу 19 и будут обрабатываться так же, как команда ADD с двумя регистрами в качестве операндов. Класс команд обозначает процедуру, которая вызывается для обработки всех команд данного типа.

В некоторых ассемблерах можно писать команды с применением непосредственной адресации, даже если соответствующей команды не существует в выходном языке. Такие команды с «псевдонепосредственными» адресами обрабатываются следующим образом. Ассемблер назначает участок памяти для непосредственного операнда в конце программы и порождает команду, которая обращается к нему. Например, универсальная вычислительная машина IBM 3090 не имеет команд с непосредственными адресами. Тем не менее программист может написать команду

```
L 14,=F'5'
```

для загрузки в регистр 14 константы 5 размером в полное слово. Таким образом, программисту не нужно писать директиву, чтобы разместить слово в памяти, придать ему значение 5, дать ему метку, а затем использовать эту метку в команде L. Константы, для которых ассемблер автоматически резервирует память, называются **литералами**. Литералы упрощают читаемость и понимание программы, делая значение константы очевидным в исходном операторе. При первом проходе ассемблер должен создать таблицу из всех литералов, которые используются в программе. Все три компьютера, которые мы взяли в качестве примеров, содержат команды с непосредственными адресами, поэтому их ассемблеры не обеспечивают литералы. Команды с непосредственными адресами в настоящее время считаются обычными, но раньше они рассматривались как нечто совершенно необычное. Вероятно, широкое распространение литералов внушило разработчикам, что непосредственная адресация — это очень хорошая идея. Если нужны литералы, то во время ассемблирования сохраняется таблица литералов, в которой появляется новый элемент всякий раз, когда встречается литерал. После первого прохода таблица сортируется и продублированные элементы удаляются.

В листинге 7.5 показана процедура, которая лежит в основе первого прохода ассемблера. Названия процедур были выбраны таким образом, чтобы была ясна их суть. Листинг 7.5 представляет собой хорошую отправную точку для изучения. Он достаточно короткий, он легок для понимания, и из него видно, каким должен быть следующий шаг — это написание процедур, которые используются в данном листинге.

Листинг 7.5. Первый проход простого ассемблера

```
public static void pass_one() {
// Эта процедура - первый проход ассемблера
    boolean more_input=true;                    //флаг, который останавливает первый проход
    String line, symbol, literal, opcode;       //поля команды
    int location_counter, length, value, type; //переменные
    final int END_STATEMENT = -2;               //сигналы конца ввода
```

```
    location_counter = 0;     //ассемблирование первой команды в ячейке 0
    initialize_tables();      //общая инициализация

while (more_input) {                  //more_input получает значение «ложь» с помощью END
       line = read_next_line();       //считывание строки
          length =0;                  //# байт в команде
          type =0;                    //тип команды

       if (line_is_not_comment(line)) {
           symbol = check_for_symbol(line);    //Содержит ли строка метку?
           if (symbol != null)                 //если да, то записывается символ и значение
              enter_new_symbol(symbol, location_counter);
           literal = check_for_literal(line); //Содержит ли строка литерал?
           if (literal != null)                //если да, то он вводится в таблицу
              enter_new_literal(literal);
                                               //Теперь определяем тип кода операции.
                                               //-1 значит недопустимый код операции.
           opcode = extract_opcode(line);      //определяем место кода операции
           type =search_opcode_table(opcode); //находим формат, например, OP REG1,REG2
           if (type < 0)                       //Если это не код операции, является
                                               //ли это директивой?
              type = search_pseudo_table(opcode);
           switch(type) {                      //определяем длину команды
              case 1:length=get_length_of_type1 (line); break;
              case 2:length=get_length_of_type2(line); break;
                                               //другие случаи
       }
    }
    write_temp_file(type, opcode, length, line);  //информация для второго прохода
    location_counter = location_counter + length; //обновление счетчика адреса команд
    if (type == END_STATEMENT) {                  //завершился ли ввод?
       more_input = false;                        //если да, то выполняем служебные действия:
       rewind_temp_for_pass_two();                //перематываем файл обратно
       sort_literal_table();                      //сортируем таблицу литералов
       remove_redundant_literals();               //и удаляем из нее дубликаты
       }
    }
}
```

Одни процедуры будут относительно короткими, например *check_for_symbol*, которая просто выдает соответствующее обозначение в виде цепочки символов, если таковое имеется, и выдает ноль, если его нет. Другие процедуры, например *get_length_of_type1* и *get_length_of_type2*, могут быть достаточно длинными и могут сами вызывать другие процедуры. Естественно, на практике типов будет не два, а больше, и это будет зависеть от языка, который ассемблируется, и от того, сколько типов команд предусмотрено в этом языке.

Структурирование программ имеет и другие преимущества помимо простоты программирования. Если ассемблер пишется группой людей, разнообразные процедуры могут быть разбиты на куски между программистами. Все подробности получения входных данных спрятаны в процедуре *read_next_line*. Если эти детали нужно изменить (например, из-за изменений в операционной системе), то это повлияет только на одну подчиненную процедуру, и никаких изменений в самой процедуре *pass_one* делать не нужно.

По мере чтения программы во время первого прохода ассемблер должен анализировать каждую строку, чтобы найти код операции (например, ADD), определить

ее тип (набор операндов) и вычислить длину команды. Эта информация понадобится при втором проходе, поэтому ее лучше записать, чтобы не анализировать строку во второй раз. Однако переписывание входного файла потребует больше операций ввода-вывода. Что лучше — увеличить количество операций ввода-вывода, чтобы меньше времени тратить на анализ строк, или сократить количество операций ввода-вывода и потратить больше времени на анализ, зависит от скорости работы центрального процессора и диска, эффективности файловой системы и некоторых других факторов. В нашем примере мы запишем временный файл, который будет содержать тип, код операции, длину и саму входную цепочку. Именно это цепочка и будет считываться при втором проходе, и читать файл по второму разу будет не нужно.

После прочтения директивы END первый проход завершается. В этот момент можно сохранить таблицу символьных имен и таблицу литералов, если это необходимо. В таблице литералов можно произвести сортировку и удалить продублированные литералы.

Второй проход

Задача второго прохода — произвести объектную программу и напечатать протокол ассемблирования (если нужно). Кроме того, при втором проходе должна выводиться информация, необходимая компоновщику для связывания процедур, которые ассемблировались в разное время, в один выполняемый файл. В листинге 7.6 показана процедура для второго прохода.

Листинг 7.6. Второй проход простого ассемблера

```
public static void pass_two() {
                                        //Эта процедура - второй проход ассемблера
    boolean morejnput = true;           //флаг, который останавливает второй проход
    String line, opcode;                //поля команды
    int location_counter, length, type; //переменные
      final int END_STATEMENT = -2;     //сигналы конца ввода
      final int MAX_CODE =16;           //максимальное количество байтов в команде
      byte code[] = new byte[MAX_CODE]; //количество байтов в команде в порожденном коде

    location_counter = 0;               //ассемблирование первой команды в адресе 0

    while (more_input) {                //more_input устанавливается на «ложь» с помощью END
          type = read_type();           //считывание поля типа следующей строки
opcode = read_opcode();                 //считывание поля кода операции следующей строки
length = read_length();                 //считывание поля длины в следующей строке
line = read_line();                     //считывание самой входной строки
if (type != 0) {                        //тип 0 указывает на строки комментария
    switch(type) {                      //порождение выходного кода

        case 1:eval_type1(opcode, length, line, code); break;
               case 2: eval_type2(opcode, length, line, code); break;
                                        //Другие случаи
           }
        }
    write_output(code);                 // запись двоичного кода
    write_listing(code, line);          // вывод на печать одной строки
    location_counter = location_counter + length; //обновление счетчика адреса команд
    if (type == END_STATEMENT) {        // завершен ли ввод?
```

```
        more_input = false;              // если да, то выполняем служебные операции
        finish_up();                     // завершение
    }
  }
}
```

Процедура второго прохода более или менее сходна с процедурой первого прохода: строки считываются по одной и обрабатываются тоже по одной. Поскольку мы записали в начале каждой строки тип, код операции и длину (во временном файле), все они считываются, и таким образом, нам не нужно проводить анализ строк во второй раз. Основная работа по порождению кода выполняется процедурами *eval_type1*, *eval_type2* и т. д. Каждая из них обрабатывает определенную модель (например, код операции и два регистра-операнда). Полученный в результате двоичный код команды сохраняется в переменной *code*. Затем совершается контрольное считывание. Желательно, чтобы процедура *write_code* просто сохраняла в буфере накопленный двоичный код и записывала файл на диск большими порциями, чтобы сократить рабочую нагрузку на диск.

Исходный оператор и выходной (объектный) код, полученный из него (в шестнадцатеричной системе), можно напечатать или поместить в буфер, чтобы напечатать потом. После переустановки счетчика адреса команды вызывается следующий оператор.

До настоящего момента предполагалось, что исходная программа не содержит никаких ошибок. Но любой человек, который когда-нибудь писал программы на каком-либо языке, знает, насколько это предположение не соответствует действительности. Наиболее распространенные ошибки приведены ниже:

1. Используемый символ не определен.
2. Символ был определен более одного раза.
3. Имя в поле кода операции не является допустимым кодом операции.
4. Код операции не снабжен достаточным количеством операндов.
5. У кода операции слишком много операндов.
6. Восьмеричное число содержит 8 или 9.
7. Недопустимое применение регистра (например, переход к регистру).
8. Отсутствует оператор `END`.

Программисты весьма изобретательны по части новых ошибок. Ошибки с неопределенным символом часто возникают из-за опечаток. Хороший ассемблер может вычислить, какой из всех определенных символов в большей степени соответствует неопределенному, и подставить его. Для исправления других ошибок ничего кардинального предложить нельзя. Лучшее, что может сделать ассемблер при обнаружении оператора с ошибкой, — это вывести сообщение об ошибке на экран и попробовать продолжить процесс ассемблирования.

Таблица символов

Во время первого прохода ассемблер аккумулирует всю информацию о символах и их значениях. Эту информацию он должен сохранить в таблице символьных имен, к которой будет обращаться при втором проходе. Таблицу символьных имен можно организовать несколькими способами. Некоторые из них мы опишем ниже.

При применении любого из этих способов мы пытаемся смоделировать **ассоциативную память**, которая представляет собой набор пар (символьное имя, значение). По имени ассоциативная память должна выдавать его значение.

Проще всего реализовать таблицу символьных имен в виде массива пар, где первый элемент является именем (или указателем на имя), а второй — значением (или указателем на него). Если нужно найти какой-нибудь символ, то таблица символьных имен просто последовательно просматривается, пока не будет найдено соответствие. Такой метод довольно легко запрограммировать, но он медленно работает, поскольку в среднем при каждом поиске придется просматривать половину таблицы.

Другой способ организации — отсортировать таблицу по именам и для поиска имен использовать алгоритм **двоичного поиска**. В соответствии с этим алгоритмом средний элемент таблицы сравнивается с символьным именем. Если нужное имя по алфавиту идет раньше среднего элемента, значит, оно находится в первой половине таблицы. Если символьное имя по алфавиту идет после среднего элемента, значит, оно находится во второй части таблицы. Если нужное имя совпадает со средним элементом, то поиск на этом завершается.

Предположим, что средний элемент таблицы не равен символу, который мы ищем. Мы уже знаем, в какой половине таблицы он находится. Алгоритм двоичного поиска можно применить к соответствующей половине. В результате мы либо получим совпадение, либо определим нужную четверть таблицы. Таким образом, в таблице из n элементов нужный символ можно найти примерно за $\log_2 n$ попыток. Очевидно, что такой алгоритм работает быстрее, чем просто последовательный просмотр таблицы, но при этом элементы таблицы нужно сохранять в алфавитном порядке.

Совершенно другой подход — **хэш-кодирование**. Для этого подхода требуется хэш-функция, которая отображает символы (имена) в целые числа в промежутке от 0 до k–1. Такой функцией может быть функция перемножения кодов ASCII всех символов в имени. Можно перемножить все коды ASCII символов с игнорированием переполнения, а затем взять значение по модулю k или разделить полученное значение на простое число. Фактически подойдет любая входная функция, которая дает равномерное распределение значений.

Символьные имена можно хранить в таблице, состоящей из k участков, от 0 до k–1. Все пары (символьное имя, значение), в которых имя соответствует i, сохраняются в связном списке, на который указывает слот i в хэш-таблице. Если в хэш-таблице содержится n символьных имен и k слотов, то в среднем длина списка будет n/k. Если мы выберем k, приблизительно равное n, то на нахождение нужного символьного имени в среднем потребуется всего один поиск. Путем корректировки k мы можем сократить размер таблицы, но при этом скорость поиска снизится. Хэш-код показан на рис. 7.1.

Связывание и загрузка

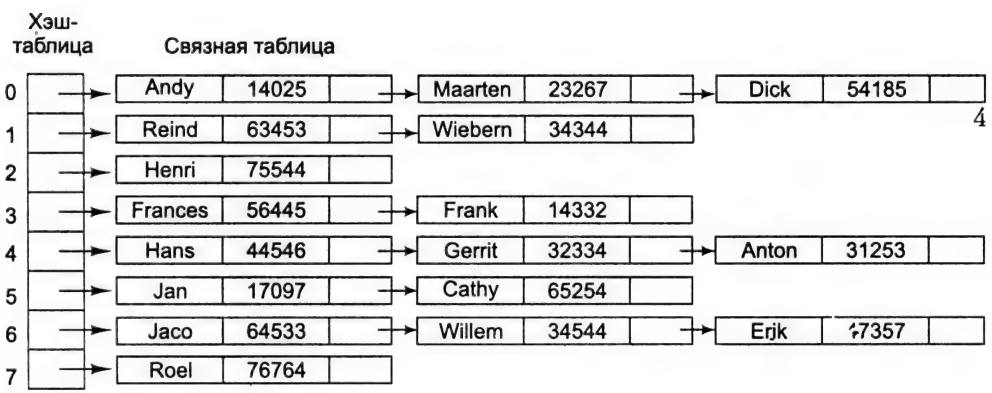
Большинство программ содержат более одной процедуры. Компиляторы и ассемблеры транслируют одну процедуру и помещают полученный на выходе результат на диск. Перед запуском программы должны быть найдены и связаны все оттран-

слированные процедуры. Если виртуальной памяти нет, связанная программа должна загружаться в основную память. Программы, которые выполняют эти функции, называются по-разному: **компоновщиками**, **связывающими загрузчиками** и **редакторами связей**. Для полной трансляции исходной программы требуется два шага, как показано на рис. 7.2:

1. Компиляция или ассемблирование исходных процедур.
2. Связывание объектных модулей.

Andy	14025	0
Anton	31253	4
Cathy	65254	5
Dick	54185	0
Erik	47357	6
Frances	56445	3
Frank	14332	3
Gerrit	32334	4
Hans	44546	4
Henri	75544	2
Jan	17097	5
Jaco	64533	6
Maarten	23267	0
Reind	63453	1
Roel	76764	7
Willem	34544	6
Wiebern	34344	1

а



б

Рис. 7.1. Хэш-кодирование: символьные имена, значения и хэш-коды, образованные от символьных имен (*а*); хэш-таблица из 8 элементов со связным списком символьных имен и значений (*б*)

Первый шаг выполняется ассемблером или компилятором, а второй — компоновщиком.

Трансляция исходной процедуры в объектном модуле — это переход на другой уровень, поскольку исходный язык и выходной язык имеют разные команды и запись. Однако при связывании перехода на другой уровень не происходит, поскольку программы на входе и на выходе компоновщика предназначены для одной и той же виртуальной машины. Задача компоновщика — собрать все процедуры, которые транслировались раздельно, и связать их вместе, чтобы в результате получился **исполняемый двоичный код**. В системах MS-DOS, Windows 95/98 и NT объектные модули имеют расширение .obj, а исполняемые двоичные программы — расширение .exe. В системе UNIX объектные модули имеют расширение .o, а исполняемые двоичные программы не имеют расширения.

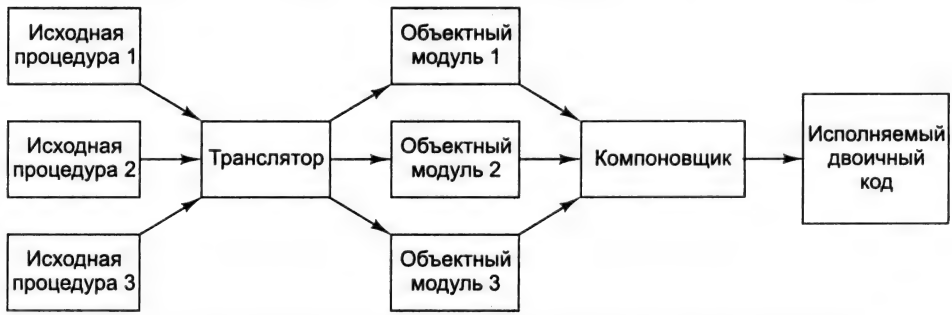


Рис. 7.2. Для получения исполняемой двоичной программы из совокупности оттранслированных независимо друг от друга процедур используется компоновщик

Компиляторы и ассемблеры транслируют каждую исходную процедуру как отдельную единицу. На это есть веская причина. Если компилятор или ассемблер считывал бы целый ряд исходных процедур и сразу переводил бы их в готовую программу на машинном языке, то при изменении одного оператора в исходной процедуре потребовалось бы заново транслировать все исходные процедуры.

Если каждая процедура транслируется по отдельности, как показано на рис. 7.2, то транслировать заново нужно будет только одну измененную процедуру, хотя понадобится заново связать все объектные модули. Однако связывание происходит гораздо быстрее, чем трансляция, поэтому выполнение этих двух шагов (трансляции и связывания) сэкономит время при доработке программы. Это особенно важно для программ, которые содержат сотни или тысячи модулей.

Задачи компоновщика

В начале первого прохода ассемблирования счетчик адреса команды устанавливается на 0. Этот шаг эквивалентен предположению, что объектный модуль во время выполнения будет находиться в ячейке с адресом 0. На рис. 7.3 показаны 4 объектных модуля для типичной машины. В этом примере каждый модуль начинается с команды перехода `BRANCH` к команде `MOVE` в том же модуле.

Чтобы запустить программу, компоновщик помещает объектные модули в основную память, формируя отображение исполняемого двоичного кода (рис. 7.4, *а*). Цель — создать точное отображение виртуального адресного пространства ис-

полняемой программы внутри компоновщика и разместить все объектные модули в соответствующих адресах. Если физической или виртуальной памяти не достаточно для формирования отображения, то можно использовать файл на диске. Обычно небольшой раздел памяти, начинающийся с нулевого адреса, используется для векторов прерывания, взаимодействия с операционной системой, обнаружения неинициализированных указателей и других целей, поэтому программы обычно начинаются не с нулевого адреса, а выше. В нашем примере программы начинаются с адреса 100.

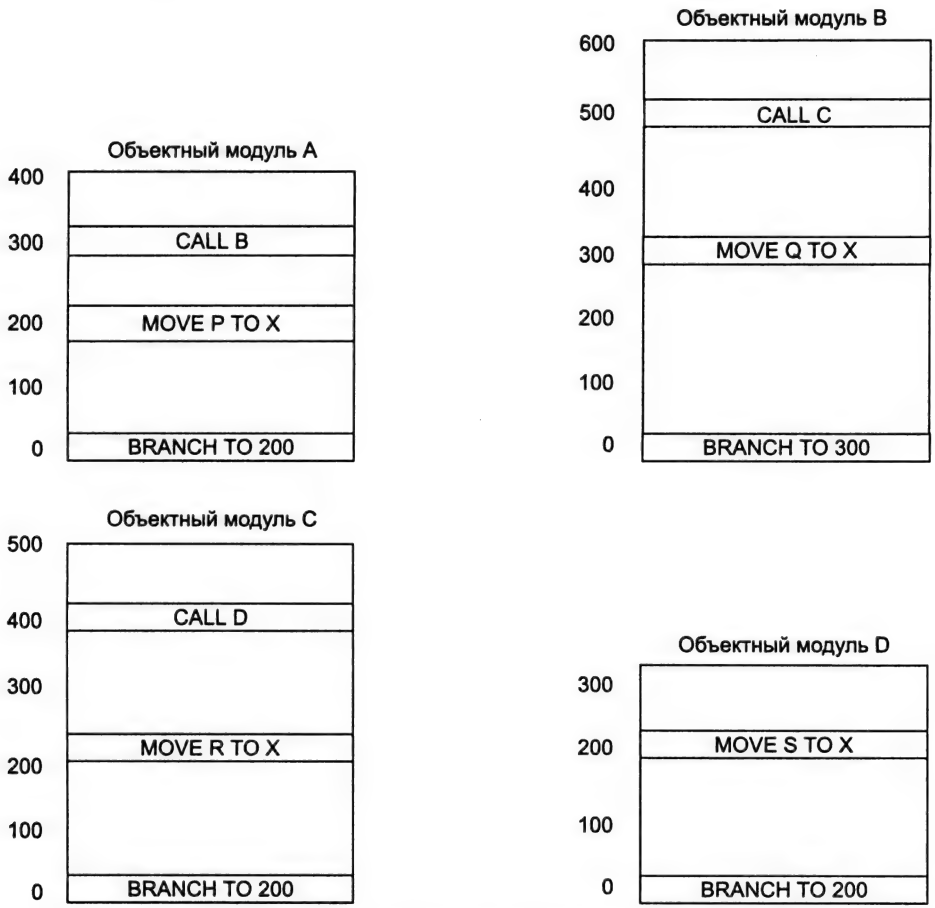


Рис. 7.3. Каждый модуль имеет свое собственное адресное пространство, начинающееся с нуля

Посмотрите на рис. 7.4, *а*. Хотя программа уже загружена в отображение исполняемого двоичного файла, она еще не готова для выполнения. Посмотрим, что произойдет, если выполнение программы начнется с команды в начале модуля A. Программа не совершит перехода к команде `MOVE`, поскольку эта команда находится в ячейке с адресом 300. Фактически все команды обращения к памяти не будут выполнены по той же причине.

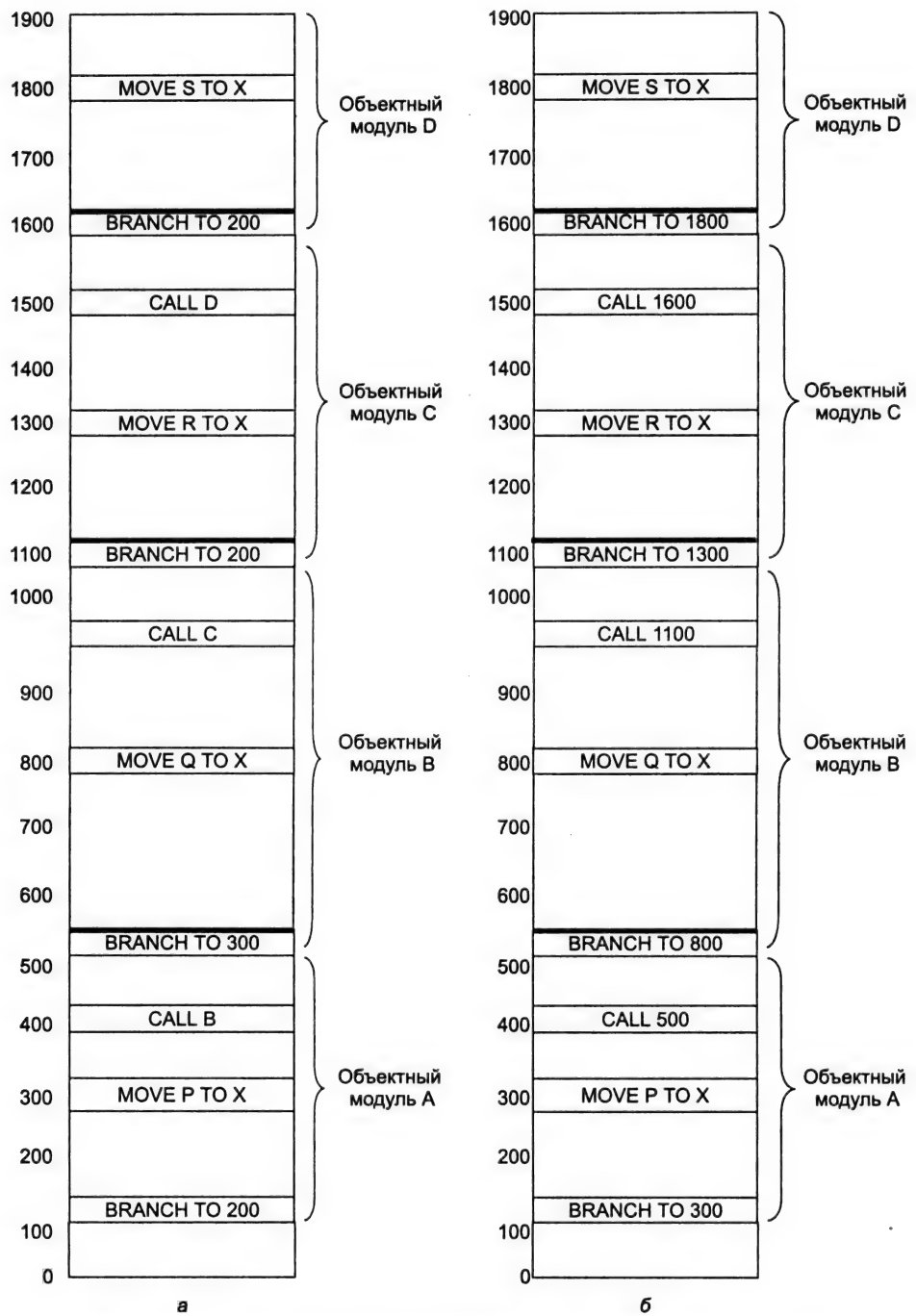


Рис. 7.4. Объектные модули после размещения в двоичном отображении, но до перераспределения памяти и связывания (*а*); те же объектные модули после связывания и перераспределения памяти (*б*). В результате получается исполняемая двоичная программа, которую можно запускать

Здесь возникает **проблема перераспределения памяти**, поскольку каждый объектный модуль на рис. 7.3 занимает отдельное адресное пространство. В машине с сегментированным адресным пространством (например, в Pentium II) каждый объектный модуль теоретически может иметь свое собственное адресное пространство, если его поместить в отдельный сегмент. Однако для Pentium II только система OS/2 поддерживает такую структуру[1]. Все версии Windows и UNIX поддерживают только одно линейное адресное пространство, поэтому все объектные модули должны быть слиты вместе в одно адресное пространство.

Более того, команды вызова процедур (см. рис. 7.4, *а*) вообще не будут работать. В ячейке с адресом 400 программист намеревается вызвать объектный модуль B, но поскольку каждая процедура транслируется отдельно, ассемблер не может определить, какой адрес вставлять в команду `CALL B`. Адрес объектного модуля B не известен до времени связывания. Такая проблема называется проблемой **внешней ссылки**. Обе проблемы решаются с помощью компоновщика.

Компоновщик сливает отдельные адресные пространства объектных модулей в единое линейное адресное пространство. Для этого совершаются следующие шаги:

1. Компоновщик строит таблицу объектных модулей и их длин.
2. На основе этой таблицы он приписывает начальные адреса каждому объектному модулю.
3. Компоновщик находит все команды, которые обращаются к памяти, и прибавляет к каждой из них **константу перемещения**, которая равна начальному адресу этого модуля.
4. Компоновщик находит все команды, которые обращаются к процедурам, и вставляет в них адрес этих процедур.

Ниже показана таблица объектных модулей рис. 7.4, построенная на первом шаге. В ней дается имя, длина и начальный адрес каждого модуля.

Модуль	Длина	Начальный адрес
A	400	100
B	600	500
C	500	1100
D	300	1600

На рисунке 7.4, *б* показано, как адресное пространство выглядит после выполнения компоновщиком всех шагов.

Структура объектного модуля

Объектные модули обычно состоят из шести частей (рис. 7.5). В первой части содержится имя модуля, некоторая информация, необходимая компоновщику (например, длины различных частей модуля), а иногда дата ассемблирования.

[1] Необходимо отметить, что сегментный способ организации был использован только в первой версии OS/2, которая была 16-битовой и разрабатывалась для 286-го микропроцессора. Поэтому относить эту систему к Pentium II представляется не вполне правильно. Начиная с 1993 года все последующие версии OS/2 были 32-битовыми и, как и остальные современные операционные системы, перестали поддерживать сегментирование, а стали использовать только страничный механизм. — *Примеч. научн. ред.*



Рис. 7.5. Внутренняя структура объектного модуля

Вторая часть объектного модуля — это список символов, определенных в модуле, вместе с их значениями. К этим символам могут обращаться другие модули. Например, если модуль состоит из процедуры *bigbug*, то элемент таблицы будет содержать цепочку символов «bigbug», за которой будет следовать соответствующий адрес. Программист на языке ассемблера с помощью директивы PUBLIC указывает, какие символьные имена считаются **точками входа**.

Третья часть объектного модуля состоит из списка символьных имен, которые используются в этом модуле, но определены в других модулях. Здесь также имеется список, который показывает, какие именно символьные имена используются теми или иными машинными командами. Второй список нужен для того, чтобы компоновщик мог вставить правильные адреса в команды, которые используют внешние имена. Процедура может вызывать другие независимо транслируемые процедуры, объявив имена вызываемых процедур внешними. Программист на языке ассемблера с помощью директивы EXTERN указывает, какие символы нужно объявить **внешними**. В некоторых компьютерах точки входа и внешние ссылки объединены в одной таблице.

Третья часть объектного модуля — это машинные команды и константы. Это единственная часть объектного модуля, которая будет загружаться в память для выполнения. Остальные 5 частей используются компоновщиком, а затем отбрасываются еще до начала выполнения программы.

Пятая часть объектного модуля — это словарь перемещений. К командам, которые содержат адреса памяти, должна прибавляться константа перемещения (см. рис. 7.4). Компоновщик сам не может определить, какие слова в четвертой части содержат машинные команды, а какие — константы. Поэтому в этой таблице содержится информация о том, какие адреса нужно переместить. Это может быть битовая таблица, где на каждый бит приходится потенциально перемещаемый адрес, либо явный список адресов, которые нужно переместить.

Шестая часть содержит указание на конец модуля, а иногда — контрольную сумму для определения ошибок, сделанных во время чтения модуля, и адрес, с которого нужно начинать выполнение.

Большинству компоновщиков требуется два прохода. На первом проходе компоновщик считывает все объектные модули и строит таблицу имен и длин модулей и глобальную таблицу символов, которая состоит из всех точек входа и внешних ссылок. На втором проходе модули считываются, перемещаются в памяти и связываются.

Время принятия решения и динамическое перераспределение памяти

В мультипрограммной системе программу можно считать в основную память, запустить ее на некоторое время, записать на диск, а затем снова считать в основную память для выполнения. В большой системе с большим количеством программ трудно быть уверенным, что программа считывается каждый раз в одно и то же место в памяти.

На рис. 7.6 показано, что произойдет, если уже перемещенная программа (см. рис. 7.4, *б*) будет загружена в адрес 400, а не в адрес 100, куда ее изначально поместил компоновщик. Все адреса памяти будут неправильными. Более того, информация о перемещении уже давно удалена. Даже если эта информация была бы доступна, перемещать все адреса при каждой перекачке программы было бы неудобно.

Проблема перемещения программ, уже связанных и размещенных в памяти, близко связана с моментом времени, в который совершается финальное связывание символических имен с абсолютными адресами физической памяти. В программе содержатся символические имена для адресов памяти (например, `BR L`). Время, в которое определяется адрес в основной памяти, соответствующий `L`, называется **временем принятия решения**. Существует по крайней мере шесть вариантов для времени принятия решения относительно привязок:

1. Когда пишется программа.
2. Когда программа транслируется.
3. Когда программа компонуется, но еще до загрузки.
4. Когда программа загружается.
5. Когда загружается базовый регистр, который используется для адресации.
6. Когда выполняется команда, содержащая адрес.

Если команда, содержащая адрес памяти, перемещается после связывания, этот адрес будет неправильным (предполагается, что объект, на который происходит ссылка, тоже перемещен). Если транслятор производит исполняемый двоичный код, то связывание происходит во время трансляции и программа должна быть запущена в адресе, в котором этого ожидает транслятор. При применении метода, описанного в предыдущем разделе, во время связывания символические имена соотносятся с абсолютными адресами, и именно по этой причине перемещать программы после связывания нельзя (см. рис. 7.6).

Здесь возникают два вопроса. Первый — когда символические имена связываются с виртуальными адресами, а второй — когда виртуальные адреса связываются с физическими адресами? Только после двух этих операций процесс связывания можно считать завершенным. Когда компоновщик связывает отдельные адресные пространства объектных модулей в единое линейное адресное пространство, он фактически создает виртуальное адресное пространство. Перемещение в памяти и связывание нужно для связи символических имен с определенными виртуальными адресами. Это наблюдение верно независимо от того, используется виртуальная память или нет.

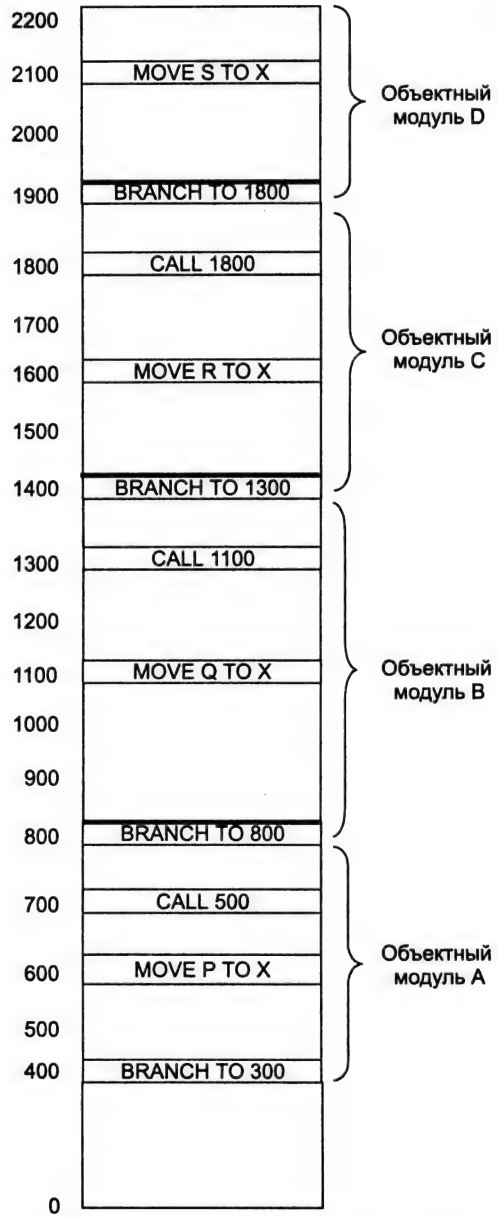


Рис. 7.6. Двоичная программа с рис. 7.4, *б*, передвинутая вверх на 300 адресов. Многие команды теперь обращаются к неправильным адресам памяти

Предположим, что адресное пространство, изображенное на рис. 7.4, *б*, было разбито на страницы. Ясно, что виртуальные адреса, соответствующие символическим именам A, B, C и D, уже определены, хотя их физические адреса будут зависеть от содержания таблицы страниц. Исполняемая двоичная программа представляет собой связывание символических имен с виртуальными адресами.

Любой механизм, который позволяет легко изменять отображение виртуальных адресов на адреса основной физической памяти, будет облегчать перемещение программы в основной памяти, даже если они уже связаны с виртуальным адресным пространством. Одним из таких механизмов является разбиение на страницы. Если программа перемещается в основной памяти, нужно изменить только ее таблицу страниц, но не саму программу.

Второй механизм — использование регистра перемещения. Компьютер CDC 6600 и его последователи содержали такой регистр. В машинах, в которых используется эта технология перемещения, регистр всегда указывает на физический адрес начала текущей программы. Аппаратное обеспечение прибавляет регистр перемещения ко всем адресам памяти, прежде чем отправить их в память. Весь процесс перемещения является «прозрачным» для каждой пользовательской программы. Пользовательские программы даже не подозревают, что этот процесс происходит. Если программа перемещается, операционная система должна обновить регистр перемещения. Такой механизм менее обычен, чем разбиение на страницы, поскольку перемещаться должна вся программа целиком (однако если есть отдельные регистры для перемещения кода и перемещения данных, как, например, в процессоре Intel 8088, то в этом случае программу нужно перемещать как два компонента).

Третий механизм можно использовать в машинах, которые могут обращаться к памяти относительно счетчика команд. Всякий раз, когда программа перемещается в основной памяти, нужно обновлять только счетчик команд. Программа, все обращения к памяти которой либо связаны со счетчиком команд, либо абсолютны (например, обращения к регистрам устройств ввода-вывода в абсолютных адресах), называется **позиционно-независимой программой**. Позиционно-независимую процедуру можно поместить в любом месте виртуального адресного пространства без настройки адресов.

Динамическое связывание

Стратегия связывания, которую мы обсуждали в разделе «Задачи компоновщика», имеет одну особенность: связь со всеми процедурами, нужными программе, устанавливается до начала работы программы. Однако если мы будем устанавливать все связи до начала работы программы в компьютере с виртуальной памятью, то мы не используем всех возможностей виртуальной памяти. Многие программы содержат процедуры, которые вызываются только при определенных обстоятельствах. Например, компиляторы содержат процедуры для компиляции редко используемых операторов, а также процедуры для исправления ошибок, которые встречаются редко.

Более гибкий способ связывания отдельно скомпилированных процедур — установление связи с каждой процедурой в тот момент, когда она впервые вызывается. Этот процесс называется **динамическим связыванием**. Впервые он был применен в системе MULTICS. В следующих разделах мы рассмотрим применение динамического связывания в нескольких системах.

Динамическое связывание в системе MULTICS

В системе MULTICS с каждой программой соотносится сегмент, так называемый **сегмент связи**, содержащий один блок информации для каждой процедуры, кото-

рая может быть вызвана. Этот блок информации начинается со слова, зарезервированного для виртуального адреса процедуры, а за ним следует имя процедуры, которое сохраняется в виде цепочки символов.

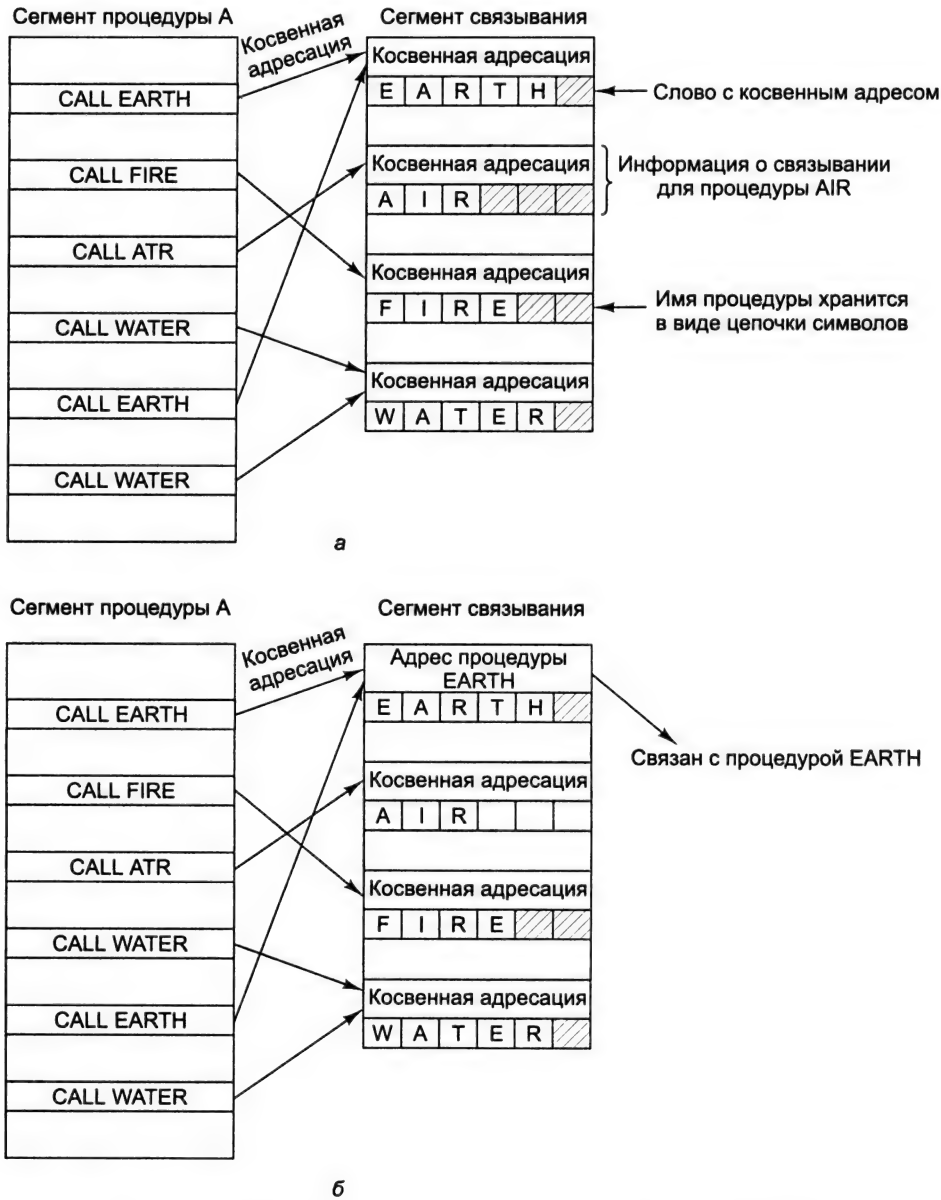


Рис. 7.7. Динамическое связывание: до вызова процедуры EARTH (*а*); после того как процедура EARTH была вызвана и связана (*б*)

При применении динамического связывания вызовы процедур во входном языке транслируются в команды, которые с помощью косвенной адресации обраща-

ются к первому слову соответствующего блока, как показано на рис. 7.7, *а*. Компилятор заполняет это слово либо недействительным адресом, либо специальным набором битов, который вызывает системное прерывание (ловушку).

Когда вызывается процедура в другом сегменте, попытка косвенно обратиться к недействительному слову вызывает системное прерывание компоновщика. Затем компоновщик находит цепочку символов в слове, которое следует за недействительным адресом, и начинает искать пользовательскую директорию для скомпилированной процедуры с таким именем. Затем этой процедуре приписывается виртуальный адрес (обычно в ее собственном сегменте), и этот виртуальный адрес записывается поверх недействительного адреса, как показано на рис. 7.7, *б*. После этого команда, которая вызвала ошибку, выполняется заново, что позволяет программе продолжать работу с того места, где она находилась до системного прерывания.

Все последующие обращения к этой процедуре будут выполняться без ошибок, поскольку слово с косвенным адресом теперь содержит действительный виртуальный адрес. Следовательно, компоновщик вызывается только тогда, когда некоторая процедура вызывается впервые. После этого вызывать компоновщик уже не нужно.

Динамическое связывание в системе Windows

Все версии операционной системы Windows, в том числе NT, поддерживают динамическое связывание. При динамическом связывании используется специальный файловый формат, который называется **DLL (Dynamic Link Library — динамически подключаемая библиотека)**. Динамически подключаемые библиотеки могут содержать процедуры, данные или и то и другое вместе. Обычно они используются для того, чтобы два и более процессов могли разделять процедуры и данные библиотеки. Большинство файлов DDL имеют расширение .dll, но встречаются и другие расширения, например .drv (для библиотек драйверов — driver libraries) и .fon (для библиотек шрифтов — font libraries).

Самая распространенная форма динамически подключаемой библиотеки — библиотека, состоящая из набора процедур, которые могут загружаться в память и к которым имеют доступ несколько процессов одновременно. На рис. 7.8 показаны два процесса, которые разделяют файл DLL, содержащий 4 процедуры, A, B, C и D. Программа 1 использует процедуру A; программа 2 использует процедуру C, хотя они вполне могли бы использовать одну и ту же процедуру.

Файл DLL строится компоновщиком из коллекции входных файлов. Построение файла DDL очень похоже на построение исполняемого двоичного кода, только при создании файла DLL компоновщику передается специальный флаг, который сообщает ему, что создается именно файл DLL. Файлы DLL обычно конструируются из набора библиотечных процедур, которые могут понадобиться нескольким процессорам. Типичными примерами файлов DLL являются процедуры сопряжения с библиотекой системных вызовов Windows и большими графическими библиотеками. Применяя файлы DDL, мы можем сэкономить пространство в памяти и на диске. Если какая-то библиотека была связана с каждой программой, использующей ее, то она будет появляться во многих исполняемых двоичных программах в памяти и на диске, а забивать пространство такими дубликатами неэко-

номно. Если мы будем использовать файлы DLL, то каждая библиотека будет появляться один раз на диске и один раз в памяти.

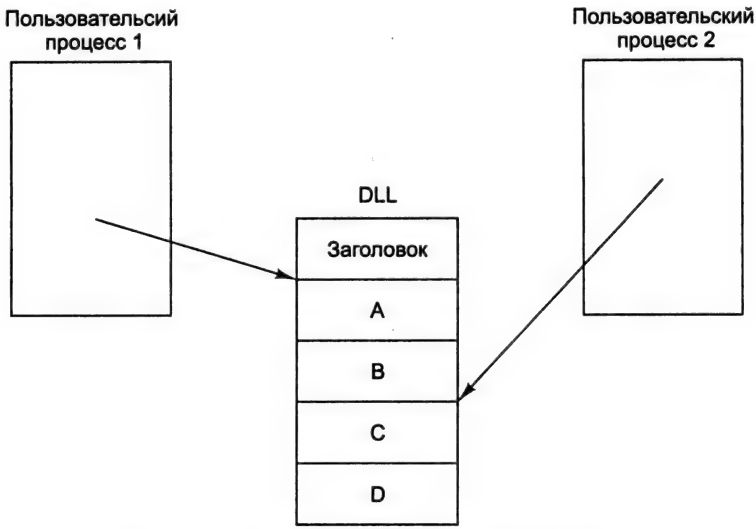


Рис. 7.8. Два процесса используют один файл DLL

Этот подход, кроме того, упрощает обновление библиотечных процедур и позволяет осуществлять обновление процедур, даже после того как программы, использующие их, были скомпилированы и связаны. Для коммерческих программных пакетов, где пользователям обычно недоступна входная программа, использование файлов DLL означает, что поставщик программного обеспечения может обнаруживать ошибки в библиотеках и исправлять положение, просто распространяя новые файлы DLL по Интернету, причем при этом не требуется производить никаких изменений в основных бинарных программах.

Основное различие между файлом DLL и исполняемой двоичной программой состоит в том, что файл DLL не может запускаться и работать сам по себе (поскольку у него нет ведущей программы). Он также содержит совершенно другую информацию в заголовке. Кроме того, файл DLL имеет несколько дополнительных процедур, не связанных с процедурами в библиотеке. Например, существует одна процедура, которая вызывается автоматически всякий раз, когда новый процесс связывается с файлом DLL, и еще одна процедура, которая вызывается автоматически всякий раз, когда связь процесса с файлом DLL отменяется. Эти процедуры могут выделять и освобождать память или управлять другими ресурсами, которые необходимы файлу DLL.

Программа может установить связь с файлом DLL двумя способами: с помощью неявного связывания и с помощью явного связывания. При **неявном связывании** пользовательская программа статически связывается со специальным файлом, так называемой **библиотекой импорта**, которая образована обслуживающей программой (утилитой), извлекающей определенную информацию из файла DLL. Библиотека импорта обеспечивает связующий элемент, который позволяет пользовательской программе получать доступ к файлу DLL. Пользовательская програм-

ма может быть связана с несколькими библиотеками импорта. Когда программа, которая применяет неявное связывание, загружается в память для выполнения, система Windows проверяет, какие файлы DLL использует эта программа и все ли эти файлы уже находятся в памяти. Те файлы, которых еще нет в памяти, загружаются туда немедленно (но необязательно целиком, поскольку они разбиты на страницы). Затем производятся некоторые изменения в структурах данных в библиотеках импорта так, чтобы можно было определить местоположение вызываемых процедур (это похоже на изменения, показанные на рис. 7.7). Их тоже нужно отобразить в виртуальное адресное пространство программы. С этого момента пользовательскую программу можно запускать. Теперь она может вызывать процедуры в файлах DLL, как будто они статически связаны с ней.

Альтернативой неявного связывания является **явное связывание**. Такой подход не требует библиотек импорта, и при нем не нужно загружать файлы DLL одновременно с пользовательской программой. Вместо этого пользовательская программа делает явный вызов прямо во время работы, чтобы установить связь с файлом DLL, а затем совершает дополнительные вызовы, чтобы получить адреса процедур, которые ей требуются. Когда все это сделано, программа совершает финальный вызов, чтобы разорвать связь с файлом DLL. Когда последний процесс разрывает связь с файлом DLL, этот файл может быть удален из памяти.

Важно осознавать, что процедура в файле DLL не имеет отличительных особенностей (как поток или процесс). Она работает в потоке вызывающей программы и для своих локальных переменных использует стек вызывающей программы. Она может содержать специфичные для процесса статические данные (а также разделенные данные) и в остальном работает как статически связанная процедура. Единственным существенным отличием является способ установления связи.

Динамическое связывание в системе UNIX

В системе UNIX используется механизм, по сути сходный с файлами DLL в Windows. Это **библиотека коллективного доступа**. Как и файл DLL, библиотека коллективного доступа представляет собой архивный файл, содержащий несколько процедур или модулей данных, которые присутствуют в памяти во время работы программы и одновременно могут быть связаны с несколькими процессами. Стандартная библиотека C и большинство сетевых программ являются библиотеками коллективного доступа.

Система UNIX поддерживает только неявное связывание, поэтому библиотека коллективного доступа состоит из двух частей: **главной библиотеки** (host library), которая статически связана с исполняемым файлом, и **целевой библиотеки** (target library), которая вызывается во время работы программы. Несмотря на некоторые различия в деталях, по существу это то же, что файлы DLL.

Краткое содержание главы

Хотя большинство программ можно и нужно писать на языках высокого уровня, существуют такие ситуации, в которых необходимо применять язык ассемблера, по крайней мере отчасти. Это программы для компьютеров с недостаточным коли-

чеством ресурсов (например, кредитные карточки, различные приборы и портативные цифровые устройства). Программа на языке ассемблера — это символическая репрезентация программы на машинном языке. Она транслируется на машинный язык специальной программой, которая называется ассемблером.

Если для успеха какого-либо аппарата требуется быстрое выполнение программы, то лучше всего сначала написать программу на языке высокого уровня, затем путем измерений установить, выполнение какой части программы занимает большую часть времени, и переписать на языке ассемблера только эту часть программы. Практика показывает, что часто небольшая часть всей программы занимает большую часть всего времени выполнения этой программы.

Во многих ассемблерах предусмотрены макросы, которые позволяют программистам давать символические имена целым последовательностям команд. Обычно эти макросы могут быть параметризированы прямым путем. Макросы реализуются с помощью алгоритма обработки строковых литералов.

Большинство ассемблеров двухпроходные. Во время первого прохода строится таблица символов для меток, литералов и объявляемых идентификаторов. Символьные имена можно либо не сортировать и искать путем последовательного просмотра таблицы, либо сначала сортировать, а потом применять двоичный поиск, либо хэшировать. Если символьные имена не нужно удалять во время первого прохода, то хэширование — это лучший метод. Во время второго прохода происходит порождение программы. Одни директивы выполняются при первом проходе, а другие — при втором.

Программы, которые ассемблируются независимо друг от друга, можно связать вместе и получить исполняемую двоичную программу, которую можно запускать. Эту работу выполняет компоновщик. Его задачи — это перемещение в памяти и связывание имен. Динамическое связывание — это технология, при которой определенные процедуры не связываются до тех пор, пока они не будут вызваны. Библиотеки коллективного пользования в системе UNIX и файлы DLL (динамически подсоединяемые библиотеки) в системе Windows используют технологию динамического связывания.

Вопросы и задания

1. 1% определенной программы отвечает за 50% времени выполнения этой программы. Сравните следующие три стратегии с точки зрения времени программирования и времени выполнения. Предположим, что для написания программы на языке C потребуется 100 человеко-месяцев, а программу на языке ассемблера написать в 10 раз труднее, но зато она работает в 4 раза эффективнее.
    1. Вся программа написана на языке C.
    2. Вся программа написана на ассемблере.
    3. Программа сначала написана на C, а затем нужный 1% программы переписан на ассемблере.
2. Для двухпроходных ассемблеров существуют определенные соглашения. Подходят ли они для компиляторов?

3. Придумайте, как программисты на языке ассемблера могут определять синонимы для кодов операций. Как это можно реализовать?
4. Все ассемблеры для процессоров Intel имеют в качестве первого операнда адрес назначения, а в качестве второго — исходный адрес. Какие проблемы могут возникнуть при другом подходе?
5. Можно ли следующую программу ассемблировать в два прохода? *Примечание*: `EQU` — это директива, которая приравнивает метку и выражение в поле операнда.

```
A EQU B
B EQU C
C EQU D
D EQU 4
```

6. Одна компания планирует разработать ассемблер для компьютера с 40-битным словом. Чтобы снизить стоимость, менеджер проекта, доктор Скрудж, решил ограничить длину символьных имен, чтобы каждое имя можно было хранить в одном слове. Скрудж объявил, что символьные имена могут состоять только из букв, причем буква Q запрещена. Какова максимальная длина символьного имени? Опишите вашу схему кодировки.
7. Чем отличается команда от директивы?
8. Чем отличается счетчик адреса команд от счетчика команд? А существует ли вообще между ними различие? Ведь и тот и другой следят за следующей командой в программе.
9. Какой будет таблица символов (имен) после обработки следующих операторов ассемблера для Pentium II (первому оператору приписан адрес 1000)?

```
EVEREST: POP BX (1 байт)
   K2: PUSH BP (1 байт)
   WHITNEY: MOV BP,SP (2 байта)
   MCKINLEY: PUSH X (3 байта)
   FUJI: PUSH SI (1 байт)
   KIBO: SUB SI,300 (3 байта)
```

10. Можете ли вы представить себе обстоятельства, при которых метка совпадет с кодом операции (например, может ли быть `MOV` меткой)? Аргументируйте.
11. Какие шаги нужно совершить, чтобы, используя двоичный поиск, найти элемент «Berkeley» в следующем списке: Ann Arbor, Berkeley, Cambridge, Eugene, Madison, New Haven, Palo Alto, Pasadena, Santa Cruz, Stony Brook, Westwood, Yellow Springs. Когда будете вычислять средний элемент в списке из четного числа элементов, возьмите элемент, который идет сразу после среднего индекса.
12. Можно ли использовать двоичный поиск в таблице, в которой содержится простое число элементов?
13. Вычислите хэш-код для каждого из следующих символьных имен. Для этого сложите буквы (A=1, B=2 и т. д.) и возьмите результат по модулю размера хэш-таблицы. Хэш-таблица содержит 19 слотов (от 0 до 18).

    els, jan, jelle, maaike

Образует ли каждое символьное имя уникальное значение хэш-функции? Если нет, то как можно разрешить эту коллизию?

14. Метод хэш-кодирования, описанный в тексте, связывает все элементы, имеющие один хэш-код, в связном списке. Альтернативный метод — иметь только одну таблицу из n слотов, в которой в каждом слоте имеется пространство для одного ключа и его значения (или для указателей на них). Если алгоритм хэширования порождает слот, который уже заполнен, производится вторая попытка с использованием того же алгоритма хэширования. Если и на этот раз слот заполнен, алгоритм используется снова и т. д. Так продолжается до тех пор, пока не будет найден пустой слот. Если доля слотов, которые уже заполнены, составляет R, сколько попыток в среднем понадобится для того, чтобы ввести в таблицу новый символ?
15. Вероятно, когда-нибудь в будущем на одну микросхему можно будет помещать тысячи идентичных процессоров, каждый из которых содержит несколько слов локальной памяти. Если все процессоры могут считывать и записывать три общих регистра, то как можно реализовать ассоциативную память?
16. Pentium II имеет сегментированную архитектуру. Сегменты независимы. Ассемблер для этой машины может содержать директиву `SEG N`, которая помещает последующий код и данные в сегмент N. Повлияет ли такая схема на счетчик адреса команды?
17. Программы часто связаны с многочисленными файлами DLL (динамически подсоединяемыми библиотеками). А не будет ли более эффективным просто поместить все процедуры в один большой файл DLL, а затем установить связь с ним?
18. Можно ли отобразить файл DLL в виртуальные адресные пространства двух процессов с разными виртуальными адресами? Если да, то какие проблемы при этом возникают? Можно ли их разрешить? Если нет, то что можно сделать, чтобы устранить их?
19. Опишем один из способов связывания. Перед сканированием библиотеки компоновщик составляет список необходимых процедур, то есть имен, которые в связываемых модулях определены как внешние (EXTERN). Затем компоновщик последовательно просматривает всю библиотеку, извлекая каждую процедуру, которая находится в списке нужных имен. Будет ли работать такая схема? Если нет, то почему, и как это можно исправить?
20. Может ли регистр использоваться в качестве фактического параметра в макровызове? А константа? Если да, то почему. Если нет, то почему.
21. Вам нужно реализовать макроассемблер. Из эстетических соображений ваш начальник решил, что макроопределения не должны предшествовать вызовам макросов. Как повлияет это решение на реализацию?
22. Подумайте, как можно поместить макроассемблер в бесконечный цикл.
23. Компоновщик считывает 5 модулей, длины которых составляют 200, 800, 600, 500 и 700 слов соответственно. Если они загружаются в этом порядке, то каковы константы перемещения?

24. Напишите модуль таблицы символов, состоящий из двух процедур: *enter*(*symbol*, *value*) и *lookup*(*symbol*, *value*). Первый вводит новые символьные имена в таблицу, а второй ищет их в таблице. Используйте какую-либо хэш-кодировку.
25. Напишите простой ассемблер для компьютера Mic-1, о котором мы говорили в главе 4. Помимо оперирования машинными командами обеспечьте возможность приписывать константы символьным именам во время ассемблирования, а также способ ассемблировать константу в машинное слово.
26. Добавьте макросы к ассемблеру, который вы должны были написать, выполняя предыдущее задание.

Глава 8
Архитектуры компьютеров параллельного действия

Скорость работы компьютеров становится все выше, но и требования, предъявляемые к ним, тоже постоянно растут. Астрономы хотят воспроизвести всю историю Вселенной с момента большого взрыва до самого конца. Фармацевты хотели бы разрабатывать новые лекарственные препараты для конкретных заболеваний с помощью компьютеров, не принося в жертву целые легионы крыс. Разработчики летательных аппаратов могли бы прийти к более эффективным результатам, если бы всю работу за них выполняли компьютеры, и тогда им не нужно было бы конструировать аэродинамическую трубу. Если говорить коротко, насколько бы мощными ни были компьютеры, этого никогда не хватает для решения многих задач (особенно научных, технических и промышленных).

Скорость работы тактовых генераторов постоянно повышается, но скорость коммутации нельзя увеличивать бесконечно. Главной проблемой остается скорость света, и заставить протоны и электроны перемещаться быстрее невозможно. Из-за высокой теплоотдачи компьютеры превратились в кондиционеры. Наконец, поскольку размеры транзисторов постоянно уменьшаются, в какой-то момент времени каждый транзистор будет состоять из нескольких атомов, поэтому основной проблемой могут стать законы квантовой механики (например, гейзенберговский принцип неопределенности).

Чтобы решать более сложные задачи, разработчики обращаются к компьютерам параллельного действия. Невозможно построить компьютер с одним процессором и временем цикла в 0,001 нс, но зато можно построить компьютер с 1000 процессорами, время цикла каждого из которых составляет 1 нс. И хотя во втором случае мы используем процессоры, которые работают с более низкой скоростью, общая производительность теоретически должна быть такой же.

Параллелизм можно вводить на разных уровнях. На уровне команд, например, можно использовать конвейеры и суперскалярную архитектуру, что позволяет увеличивать производительность примерно в 10 раз. Чтобы увеличить производительность в 100, 1000 или 1 000 000 раз, нужно продублировать процессор или, по крайней мере, какие-либо его части и заставить все эти процессоры работать вместе.

В этой главе мы изложим основные принципы разработки компьютеров параллельного действия и рассмотрим различные примеры. Все эти машины состоят из элементов процессора и элементов памяти. Отличаются они друг от друга количеством элементов, их типом и способом взаимодействия между элементами. В одних

разработках используется небольшое число очень мощных элементов, а в других — огромное число элементов со слабой мощностью. Существуют промежуточные типы компьютеров. В области параллельной архитектуры проведена огромная работа. Мы кратко расскажем об этом в данной главе. Дополнительную информацию можно найти в книгах [86, 115, 131, 159].

Вопросы разработки компьютеров параллельного действия

Когда мы сталкиваемся с новой компьютерной системой параллельного действия, возникает три вопроса:

1. Каков тип, размер и количество процессорных элементов?
2. Каков тип, размер и количество модулей памяти?
3. Как взаимодействуют элементы памяти и процессорные элементы?

Рассмотрим каждый из этих пунктов. Процессорные элементы могут быть самых различных типов — от минимальных АЛУ до полных центральных процессоров, а по размеру один элемент может быть от небольшой части микросхемы до кубического метра электроники. Очевидно, что если процессорный элемент представляет собой часть микросхемы, то можно поместить в компьютер огромное число таких элементов (например, миллион). Если процессорный элемент представляет собой целый компьютер со своей памятью и устройствами ввода-вывода, цифры будут меньше, хотя были сконструированы такие системы даже с 10 000 процессорами. Сейчас компьютеры параллельного действия конструируются из серийно выпускаемых частей. Разработка компьютеров параллельного действия часто зависит от того, какие функции выполняют эти части и каковы ограничения.

Системы памяти часто разделены на модули, которые работают независимо друг от друга, чтобы несколько процессоров одновременно могли осуществлять доступ к памяти. Эти модули могут быть маленького размера (несколько килобайтов) или большого размера (несколько мегабайтов). Они могут находиться или рядом с процессорами, или на другой плате. Динамическая память (динамическое ОЗУ) работает гораздо медленнее центральных процессоров, поэтому для повышения скорости доступа к памяти обычно используются различные схемы кэш-памяти. Может быть два, три и даже четыре уровня кэш-памяти.

Хотя существуют самые разнообразные процессоры и системы памяти, системы параллельного действия различаются в основном тем, как соединены разные части. Схемы взаимодействия можно разделить на две категории: статические и динамические. В статических схемах компоненты просто связываются друг с другом определенным образом. В качестве примеров статических схем можно привести звезду, кольцо и решетку. В динамических схемах все компоненты подсоединены к переключательной схеме, которая может трассировать сообщения между компонентами. У каждой из этих схем есть свои достоинства и недостатки.

Компьютеры параллельного действия можно рассматривать как набор микросхем, которые соединены друг с другом определенным образом. Это один подход. При другом подходе возникает вопрос, какие именно процессы выполняются

параллельно. Здесь существует несколько вариантов. Некоторые компьютеры параллельного действия одновременно выполняют несколько независимых задач. Эти задачи никак не связаны друг с другом и не взаимодействуют. Типичный пример — компьютер, содержащий от 8 до 64 процессоров, представляющий собой большую систему UNIX с разделением времени, с которой могут работать тысячи пользователей. В эту категорию попадают системы обработки транзакций, которые используются в банках (например, банковские автоматы), на авиалиниях (например, системы резервирования) и в больших web-серверах. Сюда же относятся независимые прогоны моделирующих программ, при которых используется несколько наборов параметров.

Другие компьютеры параллельного действия выполняют одну задачу, состоящую из нескольких параллельных процессов. В качестве примера рассмотрим программу игры в шахматы, которая анализирует данные позиции на доске, порождает список возможных из этих позиций ходов, а затем порождает параллельные процессы, чтобы проанализировать каждую новую ситуацию параллельно. Здесь параллелизм нужен не для того, чтобы обслуживать большое количество пользователей, а чтобы ускорить решение одной задачи.

Далее идут машины с высокой степенью конвейеризации или с большим количеством АЛУ, которые обрабатывают одновременно один поток команд. В эту категорию попадают суперкомпьютеры со специальным аппаратным обеспечением для обработки векторных данных. Здесь решается одна главная задача, и при этом все части компьютера работают вместе над одним аспектом этой задачи (например, разные элементы двух векторов суммируются параллельно).

Эти три примера различаются по так называемой **степени детализации**. В многопроцессорных системах с разделением времени блок параллелизма достаточно велик — целая пользовательская программа. Параллельная работа больших частей программного обеспечения практически без взаимодействия между этими частями называется **параллелизмом на уровне крупных структурных единиц**. Диаметрально противоположный случай (при обработке векторных данных) называется **параллелизмом на уровне мелких структурных единиц**.

Термин «степень детализации» применяется по отношению к алгоритмам и программному обеспечению, но у него есть прямой аналог в аппаратном обеспечении. Системы с небольшим числом больших процессоров, которые взаимодействуют по схемам с низкой скоростью передачи данных, называются **системами с косвенной (слабой) связью**. Им противопоставляются **системы с непосредственной (тесной) связью**, в которых компоненты обычно меньше по размеру, расположены ближе друг к другу и взаимодействуют через специальные коммуникационные сети с высокой пропускной способностью. В большинстве случаев задачи с параллелизмом на уровне крупных структурных единиц лучше всего решаются в системах со слабой связью, а задачи с параллелизмом на уровне мелких структурных единиц лучше всего решаются в системах с непосредственной связью. Однако существует множество различных алгоритмов и множество разнообразного программного и аппаратного обеспечения. Разнообразие степени детализации и возможности различной степени связности систем привели к многообразию архитектур, которые мы будем изучать в этой главе.

В следующих разделах мы рассмотрим некоторые вопросы разработки компьютеров параллельного действия. Мы начнем с информационных моделей и сетей межсоединений, затем рассмотрим вопросы, связанные с производительностью и программным обеспечением, и, наконец, перейдем к классификации архитектур компьютеров параллельного действия.

Информационные модели

В любой системе параллельной обработки процессоры, выполняющие разные части одной задачи, должны как-то взаимодействовать друг с другом, чтобы обмениваться информацией. Как именно должен происходить этот обмен? Было предложено и реализовано две разработки: мультипроцессоры и мультикомпьютеры. Мы рассмотрим их ниже.

Мультипроцессоры

В первой разработке все процессоры разделяют общую физическую память, как показано на рис. 8.1, *а*. Такая система называется **мультипроцессором** или **системой с совместно используемой памятью**.

Мультипроцессорная модель распространяется на программное обеспечение. Все процессы, работающие вместе на мультипроцессоре, могут разделять одно виртуальное адресное пространство, отображенное в общую память. Любой процесс может считывать слово из памяти или записывать слово в память с помощью команд LOAD и STORE. Больше ничего не требуется. Два процесса могут обмениваться информацией, если один из них будет просто записывать данные в память, а другой будет считывать эти данные.

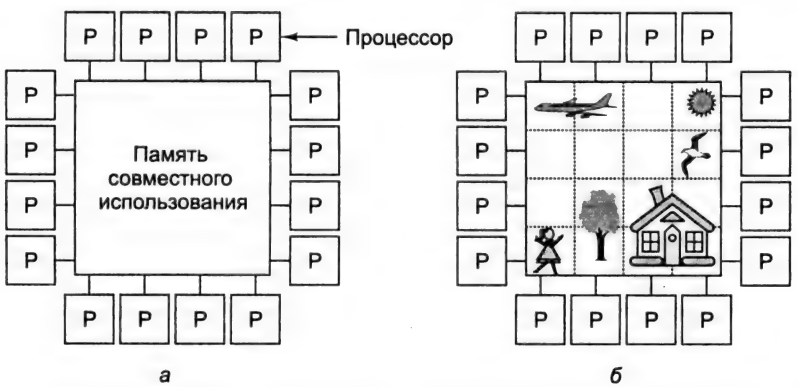


Рис. 8.1. Мультипроцессор, содержащий 16 процессоров, которые разделяют общую память (*а*); изображение, разбитое на 16 секций, каждую из которых анализирует отдельный процессор (*б*)

Благодаря такой возможности взаимодействия двух и более процессов мультипроцессоры весьма популярны. Данная модель понятна программистам и приложима к широкому кругу задач. Рассмотрим программу, которая изучает битовое отображение и составляет список всех его объектов. Одна копия отображения хранится в памяти, как показано на рис. 8.1, *б*. Каждый из 16 процессоров запускает

один процесс, которому приписана для анализа одна из 16 секций. Если процесс обнаруживает, что один из его объектов переходит через границу секции, этот процесс просто переходит вслед за объектом в следующую секцию, считывая слова этой секции. В нашем примере некоторые объекты обрабатываются несколькими процессами, поэтому в конце потребуется некоторая координация, чтобы определить количество домов, деревьев и самолетов.

В качестве примеров мультипроцессоров можно назвать Sun Enterprise 10000, Sequent NUMA-Q, SGI Origin 2000 и HP/Convex Exemplar.

Мультикомпьютеры

Во втором типе параллельной архитектуры каждый процессор имеет свою собственную память, доступную только этому процессору. Такая разработка называется **мультикомпьютером** или **системой с распределенной памятью**. Она изображена на рис. 8.2, *а*. Мультикомпьютеры обычно (хотя не всегда) являются системами со слабой связью. Ключевое отличие мультикомпьютера от мультипроцессора состоит в том, что каждый процессор в мультикомпьютере имеет свою собственную локальную память, к которой этот процессор может обращаться, выполняя команды LOAD и STORE, но никакой другой процессор не может получить доступ к этой памяти с помощью тех же команд LOAD и STORE. Таким образом, мультипроцессоры имеют одно физическое адресное пространство, разделяемое всеми процессорами, а мультикомпьютеры содержат отдельное физическое адресное пространство для каждого центрального процессора.

Поскольку процессоры в мультикомпьютере не могут взаимодействовать друг с другом просто путем чтения из общей памяти и записи в общую память, здесь необходим другой механизм взаимодействия. Они посылают друг другу сообщения, используя сеть межсоединений. В качестве примеров мультикомпьютеров можно назвать IBM SP/2, Intel/Sandia Option Red и Wisconsin COW.



Рис. 8.2. Мультикомпьютер, содержащий 16 процессоров, каждый из которых имеет свою собственную память (*а*); битовое отображение рис. 8.1, разделенное между 16 участками памяти (*б*)

При отсутствии памяти совместного использования в аппаратном обеспечении предполагается определенная структура программного обеспечения. В мультикомпьютере невозможно иметь одно виртуальное адресное пространство, из которого все процессы могут считывать информацию и в которое все процессы могут записывать информацию просто путем выполнения команд LOAD и STORE. Например, если процессор 0 (в верхнем левом углу) на рис. 8.1, *б* обнаруживает, что часть его объекта попадает в другую секцию, относящуюся к процессору 1, он может продолжать считывать информацию из памяти, чтобы получить хвост самолета. Однако если процессор 0 на рис. 8.2, *б* обнаруживает это, он не может просто считать информацию из памяти процессора 1. Для получения необходимых данных ему нужно сделать что-то другое.

В частности, ему нужно как-то определить, какой процессор содержит необходимые ему данные, и послать этому процессору сообщение с запросом копии данных. Затем процессор 0 блокируется до получения ответа. Когда процессор 1 получает сообщение, программное обеспечение должно проанализировать его и отправить назад необходимые данные. Когда процессор 0 получает ответное сообщение, программное обеспечение разблокируется и продолжает работу.

В мультикомпьютере для взаимодействия между процессорами часто используются примитивы send и receive. Поэтому программное обеспечение мультикомпьютера имеет более сложную структуру, чем программное обеспечение мультипроцессора. При этом основной проблемой становится правильное разделение данных и разумное их размещение. В мультипроцессоре размещение частей не влияет на правильность выполнения задачи, хотя может повлиять на производительность. Таким образом, мультикомпьютер программировать гораздо сложнее, чем мультипроцессор.

Возникает вопрос: зачем вообще создавать мультикомпьютеры, если мультипроцессоры гораздо проще запрограммировать? Ответ прост: гораздо проще и дешевле построить большой мультикомпьютер, чем мультипроцессор с таким же количеством процессоров. Реализация общей памяти, разделяемой несколькими сотнями процессоров, — это весьма сложная задача, а построить мультикомпьютер, содержащий 10 000 процессоров и более, довольно легко.

Таким образом, мы сталкиваемся с дилеммой: мультипроцессоры сложно строить, но легко программировать, а мультикомпьютеры легко строить, но трудно программировать. Поэтому стали предприниматься попытки создания гибридных систем, которые относительно легко конструировать и относительно легко программировать. Это привело к осознанию того, что совместную память можно реализовывать по-разному, и в каждом случае будут какие-то преимущества и недостатки. Практически все исследования в области архитектур с параллельной обработкой направлены на создание гибридных форм, которые сочетают в себе преимущества обеих архитектур. Здесь важно получить такую систему, которая **расширяема**, то есть которая будет продолжать исправно работать при добавлении все новых и новых процессоров.

Один из подходов основан на том, что современные компьютерные системы не монолитны, а состоят из ряда уровней. Это дает возможность реализовать общую

память на любом из нескольких уровней, как показано на рис. 8.3. На рис. 8.3, *а* мы видим память совместного использования, реализованную в аппаратном обеспечении в виде реального мультипроцессора. В данной разработке имеется одна копия операционной системы с одним набором таблиц, в частности таблицей распределения памяти. Если процессу требуется больше памяти, он прерывает работу операционной системы, которая после этого начинает искать в таблице свободную страницу и отображает эту страницу в адресное пространство вызывающей программы. Что касается операционной системы, имеется единая память, и операционная система следит, какая страница в программном обеспечении принадлежит тому или иному процессу. Существует множество способов реализации совместной памяти в аппаратном обеспечении.

Второй подход — использовать аппаратное обеспечение мультикомпьютера и операционную систему, которая моделирует разделенную память, обеспечивая единое виртуальное адресное пространство, разбитое на страницы. При таком подходе, который называется **DSM (Distributed Shared Memory — распределенная совместно используемая память)** [82, 83, 84], каждая страница расположена в одном из блоков памяти (см. рис. 8.2, *а*). Каждая машина содержит свою собственную виртуальную память и собственные таблицы страниц. Если процессор совершает команду `LOAD` или `STORE` над страницей, которой у него нет, происходит прерывание операционной системы. Затем операционная система находит нужную страницу и требует, чтобы процессор, который обладает нужной страницей, преобразовал ее в исходную форму и послал по сети межсоединений. Когда страница достигает пункта назначения, она отображается в память, и выполнение прерванной команды возобновляется. По существу, операционная система просто вызывает недостающие страницы не с диска, а из памяти. Но у пользователя создается впечатление, что машина содержит общую разделенную память. DSM мы рассмотрим ниже в этой главе.

Третий подход — реализовать общую разделенную память на уровне программного обеспечения. При таком подходе абстракцию разделенной памяти создает язык программирования, и эта абстракция реализуется компилятором. Например, модель Linda основана на абстракции разделенного пространства кортежей (записей данных, содержащих наборы полей). Процессы любой машины могут взять кортеж из общего пространства или отправить его в общее пространство. Поскольку доступ к этому пространству полностью контролируется программным обеспечением (системой Linda), никакого специального аппаратного обеспечения или специальной операционной системы не требуется.

Другой пример памяти совместного использования, реализованной в программном обеспечении, — модель общих объектов в системе Orca. В модели Orca процессы разделяют объекты, а не кортежи, и могут выполнять над ними те или иные процедуры. Если процедура изменяет внутреннее состояние объекта, операционная система должна проследить, чтобы все копии этого объекта на всех машинах одновременно были изменены. И опять, поскольку объекты — это чисто программное понятие, их можно реализовать с помощью программного обеспечения без вмешательства операционной системы или аппаратного обеспечения. Модели Linda и Orca мы рассмотрим ниже в этой главе.



Рис. 8.3. Уровни, на которых можно реализовать память совместного использования: аппаратное обеспечение (а); операционная система (*б*); программное обеспечение (*в*)

Сети межсоединений

На рис. 8.2 мы показали, что мультикомпьютеры связываются через сети межсоединений. Рассмотрим их подробнее. Интересно отметить, что мультикомпьютеры и мультипроцессоры очень сходны в этом отношении, поскольку мультипроцессоры часто содержат несколько модулей памяти, которые также должны быть связаны друг с другом и с процессорами. Следовательно, многое из того, о чем мы будем говорить в этом разделе, применимо к обоим типам систем.

Основная причина сходства коммуникационных связей в мультипроцессоре и мультикомпьютере заключается в том, что в обоих случаях применяется передача сообщений. Даже в однопроцессорной машине, когда процессору нужно считать или записать слово, он устанавливает определенные линии на шине и ждет ответа. Это действие представляет собой то же самое, что и передача сообщений: инициатор посылает запрос и ждет ответа. В больших мультипроцессорах взаимодействие между процессорами и удаленной памятью почти всегда состоит в том, что процессор посылает в память сообщение, так называемый **пакет**, который запрашивает определенные данные, а память посылает процессору ответный пакет.

Сети межсоединений могут состоять максимум из пяти компонентов:

1. Центральные процессоры.
2. Модули памяти.
3. Интерфейсы.
4. Каналы связи.
5. Коммутаторы.

Процессоры и модули памяти мы уже рассматривали в этой книге и больше не будем к этому возвращаться. Интерфейсы — это устройства, которые вводят и выводят сообщения из центральных процессоров и модулей памяти. Во многих разработках интерфейс представляет собой микросхему или плату, к которой подсоединяется локальная шина каждого процессора и которая может передавать сигналы процессору и локальной памяти (если таковая есть). Часто внутри интерфейса содержится программируемый процессор со своим собственным ПЗУ, которое принадлежит только этому процессору. Обычно интерфейс способен считывать и записывать информацию в различные модули памяти, что позволяет ему перемещать блоки данных.

Каналы связи — это каналы, по которым перемещаются биты. Каналы могут быть электрическими или оптико-волоконными, последовательными (шириной 1 бит) или параллельными (шириной более 1 бита). Каждый канал связи характеризуется максимальной пропускной способностью (это максимальное число битов, которое он способен передавать в секунду). Каналы могут быть симплексными (передавать биты только в одном направлении), полудуплексными (передавать информацию в обоих направлениях, но не одновременно) и дуплексными (передавать биты в обоих направлениях одновременно).

Коммутаторы — это устройства с несколькими входными и несколькими выходными портами. Когда на входной порт приходит пакет, некоторые биты в этом пакете используются для выбора выходного порта, в который посылается пакет. Размер пакета может составлять 2 или 4 байта, но может быть и значительно больше (например, 8 Кбайт).

Сети межсоединений можно сравнить с улицами города. Улицы похожи на каналы связи. Каждая улица может быть с односторонним и двусторонним движением, она характеризуется определенной «скоростью передачи данных» (имеется в виду ограничение скорости движения) и имеет определенную ширину (число рядов). Перекрестки похожи на коммутаторы. На каждом перекрестке прибывающий пакет (пешеход или машина) выбирает, в какой выходной порт (улицу) поступить дальше в зависимости от того, каков конечный пункт назначения.

При разработке и анализе сети межсоединений важно учитывать несколько ключевых моментов. Во-первых, это топология (то есть способ расположения компонентов). Во-вторых, это то, как работает система переключения и как осуществляется связь между ресурсами. В-третьих, какой алгоритм выбора маршрута используется для доставки сообщений в пункт назначения. Ниже мы рассмотрим каждый из этих пунктов.

Топология

Топология сети межсоединений определяет, как расположены каналы связи и коммутаторы (это, например, может быть кольцо или решетка). Топологии можно изображать в виде графов, в которых дуги соответствуют каналам связи, а узлы — коммутаторам (рис. 8.4). С каждым узлом в сети (или в соответствующем графе) связан определенный ряд каналов связи. Математики называют число каналов **степенью** узла, инженеры — **коэффициентом разветвления**. Чем больше степень, тем больше вариантов маршрута и тем выше отказоустойчивость. Если каждый узел содержит k дуг и соединение сделано правильно, то можно построить сеть межсоединений так, чтобы она оставалась полносвязной, даже если k–1 каналов повреждены.

Следующее свойство сети межсоединений — это ее **диаметр**. Если расстоянием между двумя узлами мы будем считать число дуг, которые нужно пройти, чтобы попасть из одного узла в другой, то диаметром графа будет расстояние между двумя узлами, которые расположены дальше всех друг от друга. Диаметр сети определяет самую большую задержку при передаче пакетов от одного процессора к другому или от процессора к памяти, поскольку каждая пересылка через канал связи занимает определенное количество времени. Чем меньше диаметр, тем выше производительность. Также имеет большое значение среднее расстояние между двумя узлами, поскольку от него зависит среднее время передачи пакета.

Еще одно важное свойство сети межсоединений — это ее пропускная способность, то есть количество данных, которое она способна передавать в секунду. Очень важная характеристика — **бисекционная пропускная способность**. Чтобы вычислить это число, нужно мысленно разделить сеть межсоединений на две равные (с точки зрения числа узлов) несвязанные части путем удаления ряда дуг из графа. Затем нужно вычислить общую пропускную способность дуг, которые мы удалили. Существует множество способов разделения сети межсоединений на две равные части. Бисекционная пропускная способность — минимальная из всех возможных. Предположим, что бисекционная пропускная способность составляет 800 бит/с. Тогда если между двумя частями много взаимодействий, то общую пропускную способность в худшем случае можно сократить до 800 бит/с. По мнению многих разработчиков, бисекционная пропускная способность — это самая важ-

ная характеристика сети межсоединений. Часто основная цель при разработке сети межсоединений — сделать бисекционную пропускную способность максимальной.

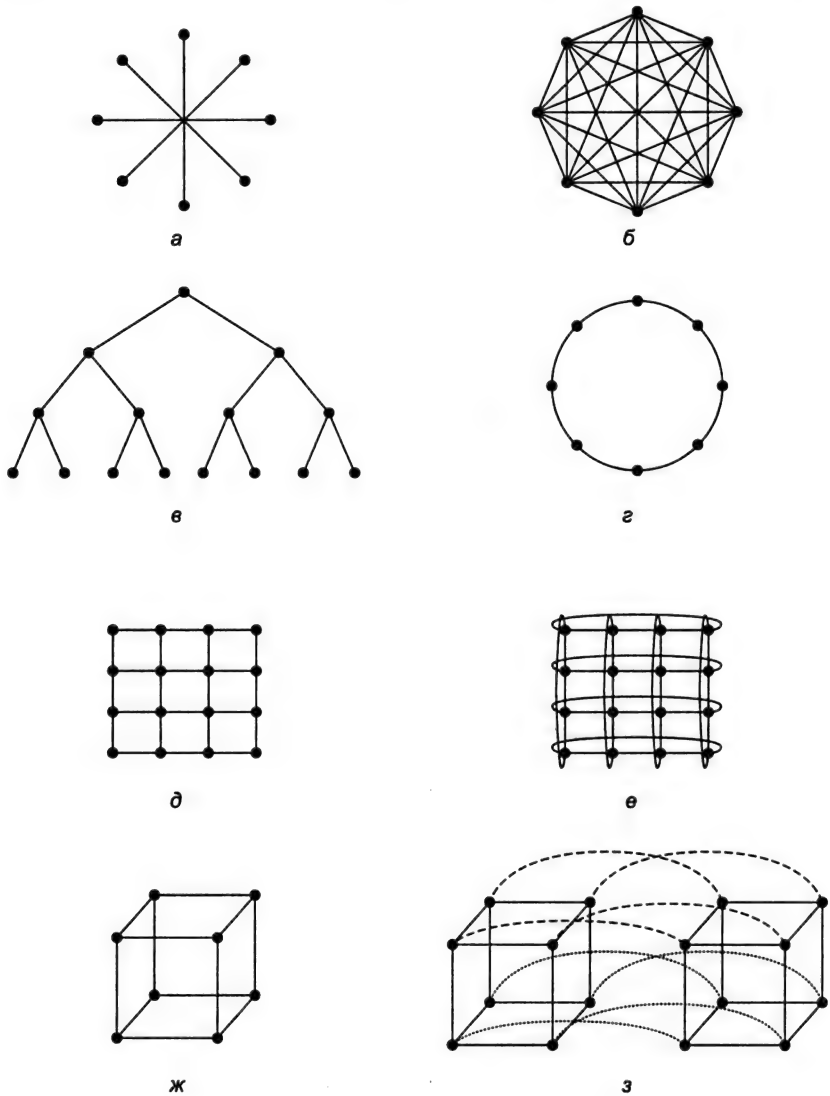


Рис. 8.4. Различные топологии. Жирные точки соответствуют коммутаторам. Процессоры и модули памяти не показаны: звезда (*а*); полное межсоединение (full interconnect) (*б*); дерево (*в*); кольцо (*г*); решетка (*д*); двойной тор (*е*); куб (*ж*); гиперкуб (*з*)

Сети межсоединений можно характеризовать по их **размерности**. Размерность определяется по числу возможных вариантов перехода из исходного пункта в пункт назначения. Если выбора нет (то есть существует только один путь из каждого исходного пункта в каждый конечный пункт), то сеть нульмерная. Если есть два возможных варианта (например, если можно пойти либо направо, либо налево),

то сеть одномерна. Если есть две оси и пакет может направиться направо или налево либо вверх или вниз, то такая сеть двумерна и т. д.

На рис. 8.4 показано несколько топологий. Здесь изображены только каналы связи (это линии) и коммутаторы (это точки). Модули памяти и процессоры (они на рисунке не показаны) подсоединяются к коммутаторам через интерфейсы. На рис. 8.4, *а* изображена нульмерная конфигурация **звезда**, где процессоры и модули памяти прикрепляются к внешним узлам, а переключение совершает центральный узел. Такая схема очень проста, но в большой системе центральный коммутатор будет главным критическим параметром, который ограничивает производительность системы. И с точки зрения отказоустойчивости это очень неудачная разработка, поскольку одна ошибка в центральном коммутаторе может разрушить всю систему.

На рис. 8.4, *б* изображена другая нульмерная топология — **полное межсоединение** (full interconnect). Здесь каждый узел непосредственно связан с каждым имеющимся узлом. В такой разработке пропускная способность между двумя секциями максимальна, диаметр минимален, а отказоустойчивость очень высока (даже при утрате шести каналов связи система все равно будет полностью взаимосвязана). Однако для k узлов требуется k(k−1)/2 каналов, а это совершенно неприемлемо для больших значений k.

На рис. 8.4, *в* изображена третья нульмерная топология — **дерево**. Здесь основная проблема состоит в том, что пропускная способность между секциями равна пропускной способности каналов. Обычно у верхушки дерева наблюдается очень большой поток обмена информации, поэтому верхние узлы становятся препятствием для повышения производительности. Можно разрешить эту проблему, увеличив пропускную способность верхних каналов. Например, самые нижние каналы будут иметь пропускную способность b, следующий уровень — пропускную способность 2b, а каждый канал верхнего уровня — пропускную способность 4b. Такая схема называется **толстым деревом (fat tree)**. Она применялась в коммерческих мультикомпьютерах Thinking Machines' CM-5.

Кольцо (рис. 8.4, *г*) — это одномерная топология, поскольку каждый отправленный пакет может пойти направо или налево. **Решетка** или **сетка** (рис. 8.4, *д*) — это двумерная топология, которая применяется во многих коммерческих системах. Она отличается регулярностью и применима к системам большого размера, а диаметр составляет квадратный корень от числа узлов (то есть при расширении системы диаметр увеличивается незначительно). **Двойной тор** (рис. 8.4, *е*) является разновидностью решетки. Это решетка, у которой соединены края. Она характеризуется большей отказоустойчивостью и меньшим диаметром, чем обычная решетка, поскольку теперь между двумя противоположными узлами всего два транзитных участка.

Куб (рис. 8.4, *ж*) — это правильная трехмерная топология. На рисунке изображен куб 2×2×2, но в общем случае он может быть k×k×k. На рис. 8.4, *з* показан четырехмерный куб, полученный из двух трехмерных кубов, которые связаны между собой. Можно сделать пятимерный куб, соединив вместе 4 четырехмерных куба. Чтобы получить 6 измерений, нужно продублировать блок из 4 кубов и соединить соответствующие узлы и т. д.; n-мерный куб называется **гиперкубом**. Эта топология используется во многих компьютерах параллельного действия, поскольку ее

диаметр находится в линейной зависимости от размерности. Другими словами, диаметр — это логарифм по основанию 2 от числа узлов, поэтому 10-мерный гиперкуб имеет 1024 узла, но диаметр равен всего 10, что дает очень незначительные задержки при передаче данных. Отметим, что решетка 32×32, которая также содержит 1024 узла, имеет диаметр 62, что более чем в шесть раз превышает диаметр гиперкуба. Однако чем меньше диаметр гиперкуба, тем больше разветвление и число каналов (и следовательно, тем выше стоимость). Тем не менее в системах с высокой производительностью чаще всего используется именно гиперкуб.

Коммутация

Сеть межсоединений состоит из коммутаторов и проводов, соединяющих их. На рисунке 8.5 изображена небольшая сеть межсоединений с четырьмя коммутаторами. В данном случае каждый коммутатор имеет 4 входных порта и 4 выходных порта. Кроме того, каждый коммутатор содержит несколько центральных процессоров и схемы соединения (на рисунке они показано не полностью). Задача коммутатора — принимать пакеты, которые приходят на любой входной порт, и отправлять пакеты из соответствующих выходных портов.

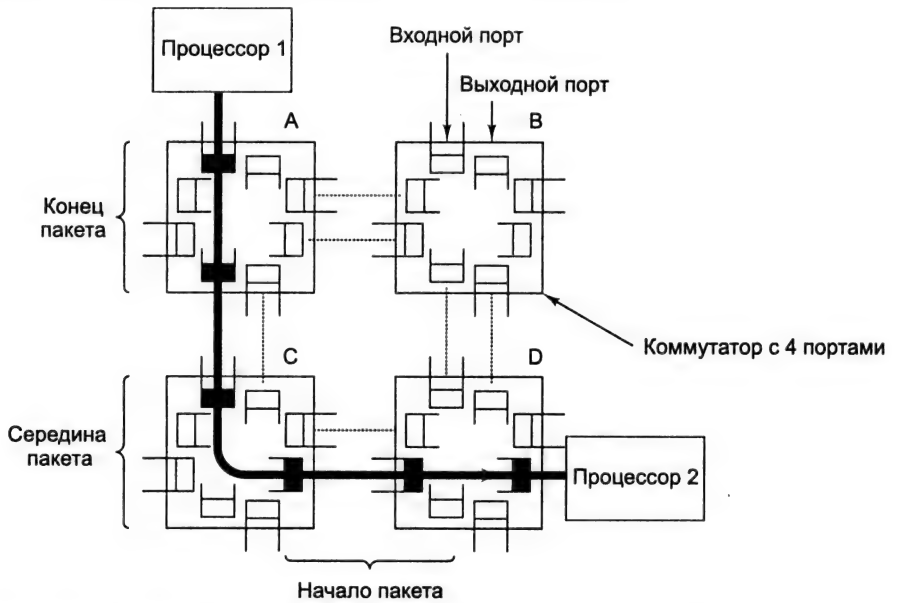


Рис. 8.5. Сеть межсоединений в форме квадратной решетки с четырьмя коммутаторами. Здесь показаны только два процессора

Каждый выходной порт связан с входным портом другого коммутатора через последовательный или параллельный канал (на рис. 8.5. это пунктирная линия). Последовательные каналы передают один бит единовременно. Параллельные каналы могут передавать несколько битов сразу. Существуют специальные сигналы для управления каналом. Параллельные каналы характеризуются более высокой производительностью, чем последовательные каналы с такой же тактовой часто-

той, но в них возникает проблема **расфазировки данных** (нужно быть уверенным, что все биты прибывают одновременно), и они стоят гораздо дороже.

Существует несколько стратегий переключения. Первая из них — **коммутация каналов.** Перед тем как послать пакет, весь путь от начального до конечного пункта резервируется заранее. Все порты и буферы затребованы заранее, поэтому когда начинается процесс передачи, все необходимые ресурсы гарантированно доступны, и биты могут на полной скорости перемещаться от исходного пункта через все коммутаторы к пункту назначения. На рис. 8.5 показана коммутация каналов, где резервируются канал от процессора 1 к процессору 2 (черная жирная стрелка). Здесь резервируются три входных и три выходных порта.

Коммутацию каналов можно сравнить с перекрытием движения транспорта во время парада, когда блокируются все прилегающие улицы. При этом требуется предварительное планирование, но после блокирования прилегающих улиц парад может продвигаться на полной скорости, поскольку никакой транспорт препятствовать этому не будет. Недостаток такого метода состоит в том, что требуется предварительное планирование и любое движение транспорта запрещено, даже если парад (или пакеты) еще не приближается.

Вторая стратегия — **коммутация с промежуточным хранением**. Здесь не требуется предварительного резервирования. Из исходного пункта посылается целый пакет к первому коммутатору, где он хранится целиком. На рис. 8.6, *а* исходным пунктом является процессор 1, а весь пакет, который направляется в процессор 2, сначала сохраняется внутри коммутатора A. Затем этот пакет перемещается в коммутатор C, как показано на рис. 8.6, *б*. Затем весь пакет целиком перемещается в коммутатор D (рис. 8.6, *в*). Наконец, пакет доходит до пункта назначения — до процессора 2. Отметим, что никакого предварительного резервирования ресурсов не требуется.

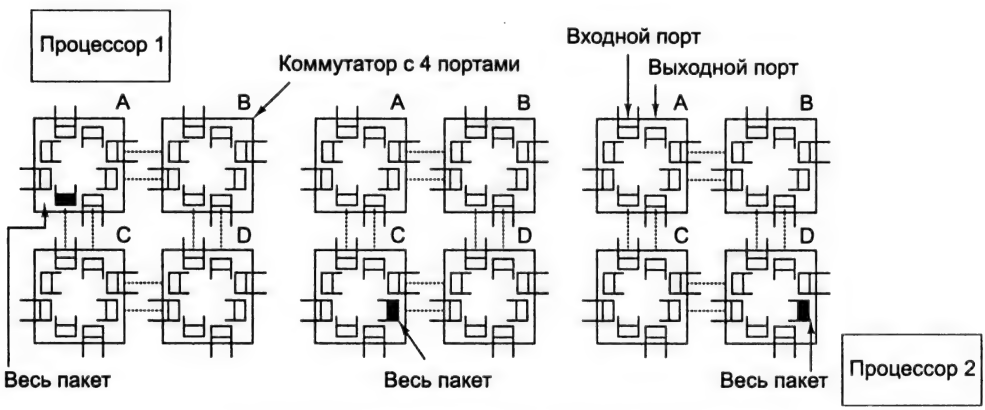


Рис. 8.6. Коммутация с промежуточным хранением

Коммутаторы с промежуточным хранением должны отправлять пакеты в буфер, поскольку когда исходный пункт (например, процессор, память или коммутатор) выдает пакет, требующийся выходной порт может быть в данный момент занят передачей другого пакета. Если бы не было буферизации, входящие пакеты, кото-

рым нужен занятый в данный момент выходной порт, пропадали бы. Применяется три метода буферизации. При **буферизации на входе** один или несколько буферов связываются с каждым входным портом в форме очереди типа FIFO («первым вошел, первым вышел»). Если пакет в начале очереди нельзя передать по причине занятости нужного выходного порта, этот пакет просто ждет своей очереди.

Однако если пакет ожидает, когда освободится выходной порт, то пакет, идущий за ним, тоже не может передаваться, даже если нужный ему порт свободен. Ситуация называется **блокировкой начала очереди**. Проиллюстрируем ситуацию на примере. Представим дорогу из двух рядов. Вереница машин в одном из рядов не может двигаться дальше, поскольку первая машина в этом ряду хочет повернуть налево, но не может из-за движения машин другого ряда. Даже если второй и всем следующим за ней машинам нужно ехать прямо, первая машина в ряду препятствует их движению.

Проблему можно устранить с помощью **буферизации на выходе**. В этой системе буферы связаны с выходными портами. Биты пакета по мере пребывания сохраняются в буфере, который связан с нужным выходным портом. Поэтому пакеты, направленные в порт m, не могут блокировать пакеты, направленные в порт n.

И при буферизации на входе, и при буферизации на выходе с каждым портом связано определенное количество буферов. Если места недостаточно для хранения всех пакетов, то какие-то пакеты придется выбрасывать. Чтобы разрешить эту проблему, можно использовать **общую буферизацию**, при которой один буферный пул динамически распределяется по портам по мере необходимости. Однако такая схема требует более сложного управления, чтобы следить за буферами, и позволяет одному занятому соединению захватить все буферы, оставив другие соединения ни с чем. Кроме того, каждый коммутатор должен вмещать самый большой пакет и даже несколько пакетов максимального размера, а для этого потребуется ужесточить требования к памяти и снизить максимальный размер пакета.

Хотя метод коммутации с промежуточным хранением гибок и эффективен, здесь возникает проблема возрастающей задержки при передаче данных по сети межсоединений. Предположим, что время, необходимое для перемещения пакета по одному транзитному участку на рис. 8.6, занимает T нс. Чтобы переместить пакет из процессора 1 в процессор 2, нужно скопировать его 4 раза (в A, в C, в D и в процессор 2), и следующее копирование не может начаться, пока не закончится предыдущее, поэтому задержка по сети составляет 4T. Чтобы выйти из этой ситуации, нужно разработать гибридную сеть межсоединений, объединяющую в себе коммутацию каналов и коммутацию пакетов. Например, каждый пакет можно разделить на части. Как только первая часть поступила в коммутатор, ее можно сразу направить в следующий коммутатор, даже если оставшиеся части пакета еще не прибыли в этот коммутатор.

Такой подход отличается от коммутации каналов тем, что ресурсы не резервируются заранее. Следовательно, возможна конфликтная ситуация в соревновании за право обладания ресурсами (портами и буферами). При **коммутации без буферизации пакетов**, если первый блок пакета не может двигаться дальше, оставшаяся часть пакета продолжает поступать в коммутатор. В худшем случае эта схема

превратится в коммутацию с промежуточным хранением. При другом типе маршрутизации, так называемой «**wormhole routing**» (**червоточина**), если первый блок не может двигаться дальше, в исходный пункт передается сигнал остановить передачу, и пакет может оборваться, будучи растянутым на два и более коммутаторов. Когда необходимые ресурсы становятся доступными, пакет может двигаться дальше.

Следует отметить, что оба подхода аналогичны конвейерному выполнению команд в центральном процессоре. В любой момент времени каждый коммутатор выполняет небольшую часть работы, и в результате получается более высокая производительность, чем если бы эту же работу выполнял один из коммутаторов.

Алгоритмы выбора маршрута

В любой сети межсоединений с размерностью один и выше можно выбирать, по какому пути передавать пакеты от одного узла к другому. Часто существует множество возможных маршрутов. Правило, определяющее, какую последовательность узлов должен пройти пакет при движении от исходного пункта к пункту назначения, называется **алгоритмом выбора маршрута**.

Хорошие алгоритмы выбора маршрута необходимы, поскольку часто свободными оказываются несколько путей. Хороший алгоритм поможет равномерно распределить нагрузку по каналам связи, чтобы полностью использовать имеющуюся в наличии пропускную способность. Кроме того, алгоритм выбора маршрута помогает избегать взаимоблокировки в сети межсоединений. Взаимоблокировка возникает в том случае, если при одновременной передаче нескольких пакетов ресурсы затребованы таким образом, что ни один из пакетов не может продвигаться дальше и все они блокируются навечно.

Пример тупиковой ситуации в сети с коммутацией каналов приведен на рис. 8.7. Тупиковая ситуация может возникать и в сети с пакетной коммутацией, но ее легче представить графически в сети с коммутацией каналов. Здесь каждый процессор пытается послать пакет процессору, находящемуся напротив него по диагонали. Каждый из них смог зарезервировать входной и выходной порты своего локального коммутатора, а также один входной порт следующего коммутатора, но он уже не может получить необходимый выходной порт на втором коммутаторе, поэтому он просто ждет, пока не освободится этот порт. Если все четыре процессора начинают этот процесс одновременно, то все они блокируются и сеть зависает.

Алгоритмы выбора маршрута можно разделить на две категории: маршрутизация от источника и распределенная маршрутизация. При **маршрутизации от источника** источник определяет весь путь по сети заранее. Этот путь выражается списком из номеров портов, которые нужно будет использовать в каждом коммутаторе по пути к пункту назначения. Если путь проходит через k коммутаторов, то первые k байтов в каждом пакете будут содержать k номеров выходных портов, 1 байт на каждый порт. Когда пакет доходит до коммутатора, первый байт отсекается и используется для определения выходного порта. Оставшаяся часть пакета затем направляется в соответствующий порт. После каждого транзитного участка пакет становится на 1 байт короче, показывая новый номер порта, который нужно выбрать в следующий раз.

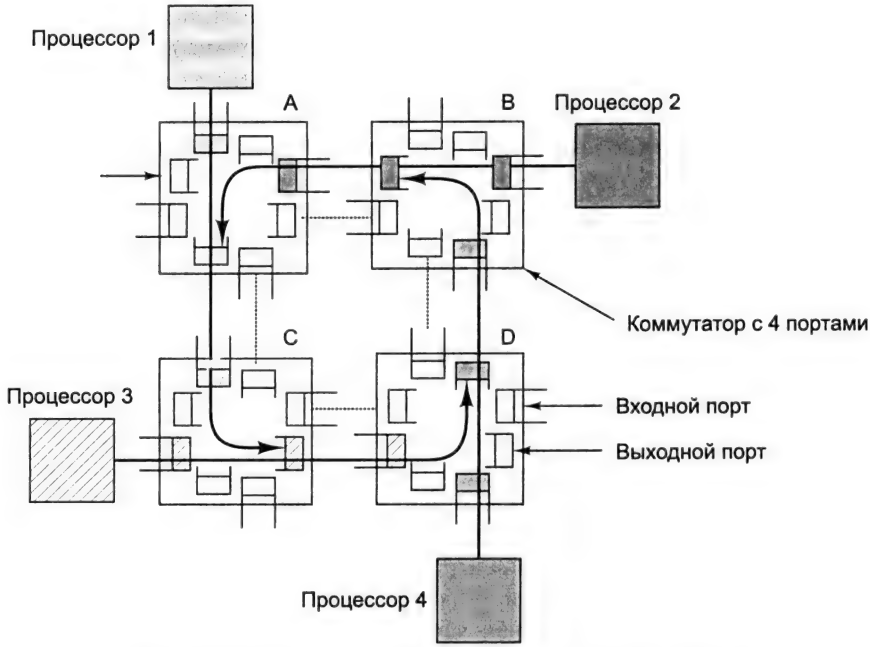


Рис. 8.7. Тупиковая ситуация в сети с коммутацией каналов

При распределенной маршрутизации каждый коммутатор сам решает, в какой порт отправить каждый приходящий пакет. Если выбор одинаков для каждого пакета, направленного к одному и тому же конечному пункту, то маршрутизация является **статической**. Если коммутатор при выборе принимает во внимание текущий трафик, то маршрутизация является **адаптивной**.

Популярным алгоритмом маршрутизации, который применяется для прямоугольных решеток с любым числом измерений и в котором никогда не возникает тупиковых ситуаций, является **пространственная маршрутизация**. В соответствии с этим алгоритмом пакет сначала перемещается вдоль оси x до нужной координаты, а затем вдоль оси y до нужной координаты и т. д. (в зависимости от количества измерений). Например, чтобы перейти из (3, 7, 5) в (6, 9, 8), пакет сначала должен переместиться из точки x=3 в точку x=6 через (4, 7, 5), (5, 7, 5) и (6, 7, 5). Затем он должен переместиться по оси y через (6, 8, 5) и (6, 9, 5). Наконец, он должен переместиться по оси z в (6, 9, 6), (6, 9, 7) и (6, 9, 8). Такой алгоритм предотвращает тупиковые ситуации.

Производительность

Цель создания компьютера параллельного действия — сделать так, чтобы он работал быстрее, чем однопроцессорная машина. Если эта цель не достигнута, то никакого смысла в построении компьютера параллельного действия нет. Более того, эта цель должна быть достигнута при наименьших затратах. Машина, которая работает в два раза быстрее, чем однопроцессорная, но стоит в 50 раз дороже послед-

ней, не будет пользоваться особым спросом. В этом разделе мы рассмотрим некоторые вопросы производительности, связанные с созданием архитектур параллельных компьютеров.

Метрика аппаратного обеспечения

В аппаратном обеспечении наибольший интерес представляет скорость работы процессоров, устройств ввода-вывода и сети. Скорость работы процессоров и устройств ввода-вывода такая же, как и в однопроцессорной машине, поэтому ключевыми параметрами в параллельной системе являются те, которые связаны с межсоединением. Здесь есть два ключевых момента: время ожидания и пропускная способность. Мы рассмотрим их по очереди.

Полное время ожидания — это время, которое требуется на то, чтобы процессор отправил пакет и получил ответ. Если пакет посылается в память, то время ожидания — это время, которое требуется на чтение и запись слова или блока слов. Если пакет посылается другому процессору, то время ожидания — это время, которое требуется на межпроцессорную связь для пакетов данного размера. Обычно интерес представляет время ожидания для пакетов минимального размера (как правило, для одного слова или небольшой строки кэш-памяти).

Время ожидания строится из нескольких факторов. Для сетей с коммутацией каналов, сетей с промежуточным хранением и сетей без буферизации пакетов характерно разное время ожидания. Для коммутации каналов время ожидания составляет сумму времени установки и времени передачи. Для установки схемы нужно выслать пробный пакет, чтобы зарезервировать необходимые ресурсы, а затем передать назад сообщение об этом. После этого можно ассемблировать пакет данных. Когда пакет готов, биты можно передавать на полной скорости, поэтому если общее время установки составляет T_s, размер пакета равен p бит, а пропускная способность b битов в секунду, то время ожидания в одну сторону составит T_s+p/b. Если схема дуплексная и никакого времени установки на ответ не требуется, то минимальное время ожидания для передачи пакета размером в p бит и получения ответа размером в p битов составляет T_s+2p/b секунд.

При пакетной коммутации не нужно посылать пробный пакет в пункт назначения заранее, но все равно требуется некоторое время установки, T_a, на компоновку пакета. Здесь время передачи в одну сторону составляет T_a+p/b, но за этот период пакет доходит только до первого коммутатора. При прохождении через сам коммутатор получается некоторая задержка, T_d, а затем происходит переход к следующему коммутатору и т. д. Время T_d состоит из времени обработки и задержки в очереди (когда нужно ждать, пока не освободится выходной порт). Если имеется n коммутаторов, то общее время ожидания в одну сторону составляет $T_a+n(p/b+T_d)+p/b$, где последнее слагаемое отражает копирование пакета из последнего коммутатора в пункт назначения.

Время ожидания в одну сторону для коммутации без буферизации пакетов и «червоточины» в лучшем случае будет приближаться к T_a+p/b, поскольку здесь нет пробных пакетов для установки схемы и нет задержки, обусловленной промежуточным хранением. По существу, это время начальной установки для компоновки пакета плюс время на передачу битов. Следовало бы еще прибавить задержку на распространение сигнала, но она обычно незначительна.

Следующая характеристика аппаратного обеспечения — пропускная способность. Многие программы параллельной обработки, особенно в естественных науках, перемещают огромное количество данных, поэтому число байтов, которое система способна перемещать в секунду, имеет очень большое значение для производительности. Существует несколько показателей пропускной способности. Один из них — пропускная способность между двумя секциями — мы уже рассмотрели. Другой показатель — **суммарная пропускная способность** — вычисляется путем суммирования пропускной способности всех каналов связи. Это число показывает максимальное число битов, которое можно передать сразу. Еще один важный показатель — средняя пропускная способность каждого процессора. Если каждый процессор способен выдавать только 1 Мбайт/с, то от сети с пропускной способностью между секциями в 100 Гбайт/с не будет толку. Скорость взаимодействия будет ограничена тем, сколько данных может выдавать каждый процессор.

На практике приблизиться к теоретически возможной пропускной способности очень трудно. Пропускная способность сокращается по многим причинам. Например, каждый пакет всегда содержит какие-то служебные сигналы и данные: это компоновка, построение заголовка, отправка. При отправке 1024 пакетов по 4 байта каждый мы никогда не достигнем той же пропускной способности, что и при отправке 1 пакета на 4096 байтов. К сожалению, для достижения маленького времени ожидания лучше использовать маленькие пакеты, поскольку большие надолго блокируют линии и коммутаторы. В результате возникает конфликт между достижением низкого времени ожидания и высокой пропускной способности. Для одних прикладных задач первое важнее, чем второе, для других — наоборот. Важно знать, что всегда можно купить более высокую пропускную способность (добавив больше проводов или поставив более широкие провода), но нельзя купить низкое время ожидания. Поэтому лучше сначала сделать время ожидания как можно меньше, а уже потом заботиться о пропускной способности.

Метрика программного обеспечения

Метрика аппаратного обеспечения показывает, на что способно аппаратное обеспечение. Но пользователей интересует совсем другое. Они хотят знать, насколько быстрее будут работать их программы на компьютере параллельного действия по сравнению с однопроцессорным компьютером. Для них ключевым показателем является коэффициент ускорения: насколько быстрее работает программа в n-процессорной системе по сравнению с 1-процессорной системой. Результаты обычно иллюстрируются графиком (рис. 8.8.). Здесь мы видим несколько разных параллельных программ, которые работают на мультикомпьютере, состоящем из 64 процессоров Pentium Pro. Каждая кривая показывает повышение скорости работы одной программы с k процессорами как функцию от k. Идеальное повышение скорости показано пунктирной линией, где использование k процессоров заставляет программу работать в k раз быстрее для любого k. Лишь немногие программы достигают совершенного повышения скорости, но есть достаточное число программ, которые приближаются к идеалу. Скорость работы N-объектной задачи с добавле-

нием новых процессоров увеличивается очень стремительно; авари (африканская игра) ускоряется вполне сносно; но инвертирование матрицы нельзя ускорить более чем в пять раз, сколько бы процессоров мы не использовали. Программы и результаты обсуждаются в книге [14].

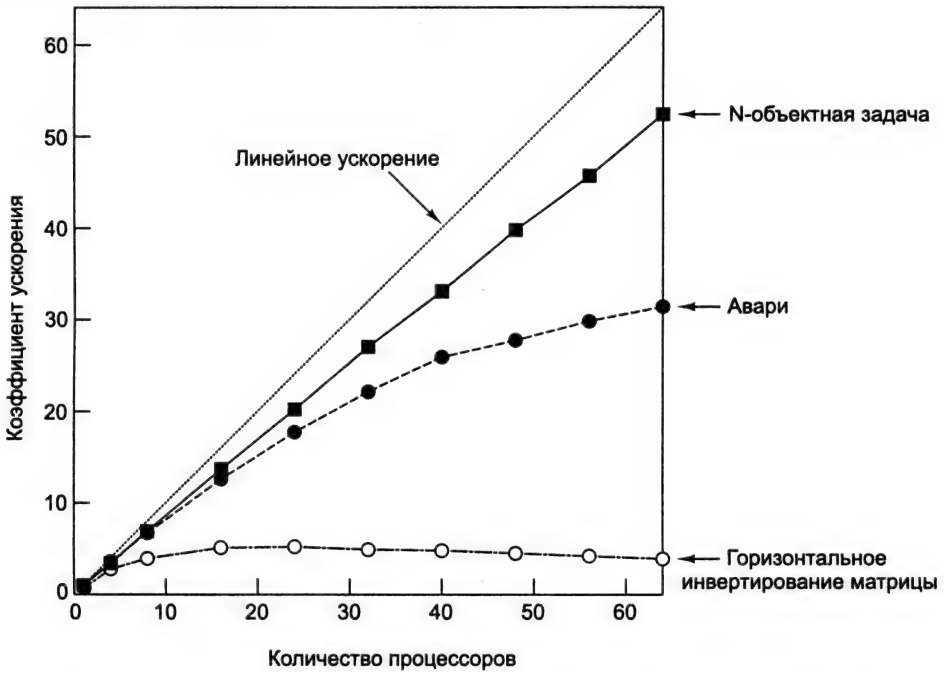


Рис. 8.8. На практике программы не могут достичь идеального повышения скорости. Идеальный коэффициент ускорения показан пунктирной линией

Есть ряд причин, по которым практически невозможно достичь идеального повышения скорости: все программы содержат последовательную часть, они часто имеют фазу инициализации, они обычно должны считывать данные и собирать результаты. Большое количество процессоров здесь не поможет. Предположим, что на однопроцессорном компьютере программа работает T секунд, причем доля (f) от этого времени является последовательным кодом, а доля (1–f) потенциально параллелизуется, как показано на рис. 8.9, а. Если второй код можно запустить на n процессорах без издержек, то время выполнения программы в лучшем случае сократится с (1–f)T до (1–f)T/n, как показано на рис. 8.9, б. В результате общее время выполнения программы (и последовательной и параллельной частей) будет fT+(1–f)T/n. Коэффициент ускорения — это время выполнения изначальной программы (T), разделенное на это новое время выполнения:

```
Speedup= n /(1+(n-1)f)
```

Для f=0 мы можем получить линейное повышение скорости, но для f>0 идеальное повышение скорости невозможно, поскольку в программе имеется последовательная часть. Это явление носит название **закона Амдала**.

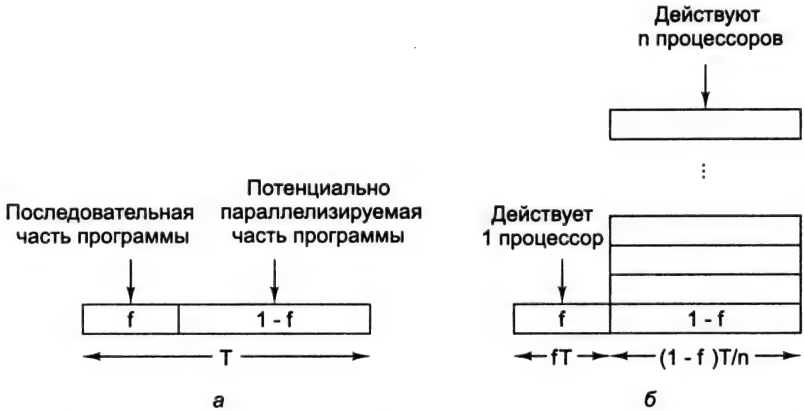


Рис. 8.9. Программа содержит последовательную часть и параллелизуемую часть (*а*); результат параллельной обработки части программы (*б*)

Закон Амдала — это только одна причина, по которой невозможно идеальное повышение скорости. Определенную роль в этом играет и время ожидания в коммуникациях, и ограниченная пропускная способность, и недостатки алгоритмов. Даже если мы имели бы в наличии 1000 процессоров, не все программы можно написать так, чтобы использовать такое большое число процессоров, а непроизводительные издержки для запуска их всех могут быть очень значительными. Кроме того, многие известные алгоритмы трудно подвергнуть параллельной обработке, поэтому в данном случае приходится использовать субоптимальный алгоритм. Для многих прикладных задач желательно заставить программу работать в n раз быстрее, даже если для этого потребуется 2n процессоров. В конце концов, процессоры не такие уж и дорогие.

Как достичь высокой производительности

Самый простой способ — включить в систему дополнительные процессоры. Однако добавлять процессоры нужно таким образом, чтобы при этом не ограничивать повышение производительности системы. Система, к которой можно добавлять процессоры и получать соответственно этому большую производительность, называется **расширяемой**.

Рассмотрим 4 процессора, которые соединены шиной (рис. 8.10, *а*). А теперь представим, что мы расширили систему до 16 процессоров, добавив еще 12 (рис. 8.10, *б*). Если пропускная способность шины составляет b Мбайт/с, то увеличив в 4 раза число процессоров, мы сократим имеющуюся пропускную способность каждого процессора с b/4 Мбайт/с до b/16 Мбайт/с. Такая система не является расширяемой.

А теперь проделаем то же действие с сеткой (решеткой) межсоединений (рис. 8.10, *в*, *г*). В такой топологии при добавлении новых процессоров мы добавляем новые каналы, поэтому при расширении системы суммарная пропускная способность на каждый процессор не снизится, как это было в случае с шиной. Отношение числа каналов к числу процессоров увеличивается от 1,0 при наличии 4 процессоров (4 процессора, 4 канала) до 1,5 при наличии 16 процессоров

(16 процессоров, 24 канала), поэтому с добавлением новых процессоров суммарная пропускная способность на каждый процессор увеличивается.

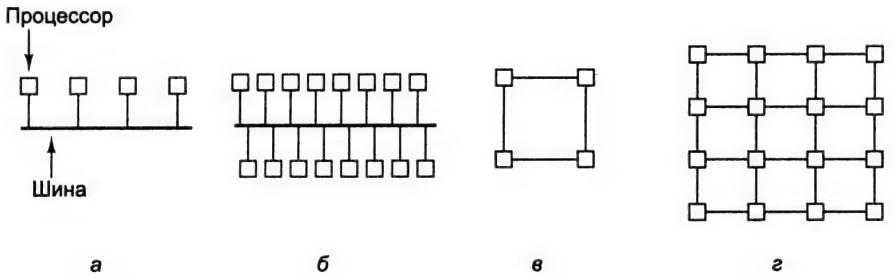


Рис. 8.10. Система из 4 процессоров, соединенных шиной (*а*); система из 16 процессоров, соединенных шиной (*б*); сетка межсоединений из 4 процессоров (*в*); сетка межсоединений из 16 процессоров (*г*)

Естественно, пропускная способность — это не единственный параметр. Добавление процессоров к шине не увеличивает диаметр сети или время ожидания при отсутствии трафика, а добавление процессоров к решетке, напротив, увеличивает. Диаметр решетки n×n равен 2(n−1), поэтому в худшем случае время ожидания растет примерно как квадратный корень от числа процессоров. Для 400 процессоров диаметр равен 38, а для 1600 процессоров — 78, поэтому если увеличить число процессоров в 4 раза, то диаметр, а следовательно, и среднее время ожидания вырастут приблизительно вдвое.

В идеале расширяемая система при добавлении новых процессоров должна сохранять одну и ту же среднюю пропускную способность на каждый процессор и постоянное среднее время ожидания. На практике сохранение достаточной пропускной способности на каждый процессор осуществимо, но время ожидания растет с увеличением размера. Лучше всего было бы сделать так, чтобы она росла логарифмически, как в гиперкубе.

Дело в том, что время ожидания часто является фатальным для производительности в мелкомодульных и среднемодульных приложениях. Если программе требуются данные, которых нет в ее локальной памяти, на их получение требуется существенное количество времени, и чем больше система, тем больше получается задержка. Эта проблема существует и в мультипроцессорах, и в мультикомпьютерах, поскольку в обоих случаях физическая память разделена на неизменяемые широко раскинувшиеся модули.

Системные разработчики применяют несколько различных технологий, которые позволяют сократить или, по крайней мере, скрыть время ожидания. Первая технология — это копирование данных. Если копии блока данных можно хранить в нескольких местах, то можно увеличить скорость доступа к этим данным. Один из возможных вариантов — использование кэш-памяти, когда одна или несколько копий блоков данных хранятся близко к тому месту, где они могут понадобиться. Другой вариант — сохранять несколько равноправных копий — копий с равным статусом (в противоположность асимметричным отношениям первичности/вторичности, которые наблюдаются при использовании кэш-памяти). Когда сохраняется несколько копий, главные вопросы — это кем, когда и куда они помещены.

Здесь возможны самые разные варианты — от динамического размещения по требованию аппаратного обеспечения до намеренного размещения во время загрузки директив компилятора. Во всех случаях главным вопросом является согласованность управления.

Вторая технология — так называемая **упреждающая выборка**. Элемент данных можно вызвать еще до того, как он понадобится. Это позволяет перекрыть процесс вызова и процесс выполнения, и когда потребуется этот элемент данных, он уже будет доступен. Упреждающая выборка может быть автоматической, а может контролироваться программой. В кэш-память загружается не только нужное слово, а вся строка кэш-памяти целиком, и другие слова из этой строки тоже могут пригодиться в будущем.

Процессом упреждающей выборки можно управлять и явным образом. Когда компилятор узнает, что ему потребуются какие-либо данные, он может выдать явную команду, чтобы получить эти данные, и выдает он эту команду заранее с таким расчетом, чтобы получить нужные данные вовремя. Такая стратегия требует, чтобы компилятор обладал полными знаниями о машине и ее синхронизации, а также контролировал, куда помещаются все данные. Спекулятивные команды LOAD работают лучше всего, когда абсолютно точно известно, что эти данные потребуются. Ошибка из-за отсутствия страницы при выполнении команды LOAD для ветви, которая в конечном итоге не используется, очень невыгодна.

Третья технология — это **многопоточная обработка**. В большинстве современных систем поддерживается мультипрограммирование, при котором несколько процессов могут работать одновременно (либо создавать иллюзию параллельной работы на основе разделения времени). Если переключение между процессами можно совершать достаточно быстро, например, предоставляя каждому из них его собственную схему распределения памяти и аппаратные регистры, то когда один процесс блокируется и ожидает прибытия данных, аппаратное обеспечение может быстро переключиться на другой процесс. В предельном случае процессор выполняет первую команду из потока 1, вторую команду из потока 2 и т. д. Таким образом, процессор всегда будет занят, даже при длительном времени ожидания в отдельных потоках.

Некоторые машины автоматически переключаются от процесса к процессу после каждой команды, чтобы скрыть длительное время ожидания. Эта идея была реализована в одном из первых суперкомпьютеров CDC 6600. Было объявлено, что он содержит 10 периферийных процессоров, которые работают параллельно. А на самом деле он содержал только один периферийный процессор, который моделировал 10 процессоров. Он выполнял по порядку по одной команде из каждого процессора, сначала одну команду из процессора 1, затем одну команду из процессора 2 и т. д.

Четвертая технология — использование неблокирующих записей. Обычно при выполнении команды STORE, процессор ждет, пока она не закончится, и только после этого продолжает работу. При наличии неблокирующих записей начинается операция памяти, но программа все равно продолжает работу. Продолжать работу программы при выполнении команды LOAD сложнее, но даже это возможно, если применять исполнение с изменением последовательности.

Программное обеспечение

Эта глава в первую очередь посвящена архитектуре параллельных компьютеров, но все же стоит сказать несколько слов о программном обеспечении. Без программного обеспечения с параллельной обработкой параллельное аппаратное обеспечение не принесет никакой пользы, поэтому хорошие разработчики аппаратного обеспечения должны учитывать особенности программного обеспечения. Подробнее о программном обеспечении для параллельных компьютеров см. [160].

Существует 4 подхода к разработке программного обеспечения для параллельных компьютеров. Первый подход — добавление специальных библиотек численного анализа к обычным последовательным языкам. Например, библиотечная процедура для инвертирования большой матрицы или для решения ряда дифференциальных уравнений с частными производными может быть вызвана из последовательной программы, после чего она будет выполняться на параллельном процессоре, а программист даже не будет знать о существовании параллелизма. Недостаток этого подхода состоит в том, что параллелизм может применяться только в нескольких процедурах, а основная часть программы останется последовательной.

Второй подход — добавление специальных библиотек, содержащих примитивы коммуникации и управления. Здесь программист сам создает процесс параллелизма и управляет им, используя дополнительные примитивы.

Следующий шаг — добавление нескольких специальных конструкций к существующим языкам программирования, позволяющих, например, легко порождать новые параллельные процессы, выполнять повторения цикла параллельно или выполнять арифметические действия над всеми элементами вектора одновременно. Этот подход широко используется, и очень во многие языки программирования были включены элементы параллелизма.

Четвертый подход — ввести совершенно новый язык специально для параллельной обработки. Очевидное преимущество такого языка — он очень хорошо подходит для параллельного программирования, но недостаток его в том, что программисты должны изучать новый язык. Большинство новых параллельных языков императивные (их команды изменяют переменные состояния), но некоторые из них функциональные, логические или объектно-ориентированные.

Существует очень много библиотек, расширений языков и новых языков, изобретенных специально для параллельного программирования, и они дают широчайший спектр возможностей, поэтому их очень трудно классифицировать. Мы сосредоточим наше внимание на пяти ключевых вопросах, которые формируют основу программного обеспечения для компьютеров параллельного действия:

1. Модели управления.
2. Степень распараллеливания процессов.
3. Вычислительные парадигмы.
4. Методы коммуникации.
5. Базисные элементы синхронизации.

Ниже мы обсудим каждый из этих вопросов в отдельности.

Модели управления

Самый фундаментальный вопрос в работе программного обеспечения — сколько будет потоков управления, один или несколько. В первой модели существует одна программа и один счетчик команд, но несколько наборов данных. Каждая команда выполняется над всеми наборами данных одновременно разными обрабатывающими элементами.

В качестве примера рассмотрим программу, которая получает ежечасные измерения температур от тысяч датчиков и должна вычислить среднюю температуру для каждого датчика. Когда программа вызывает команду

```
LOAD THE TEMPERATURE FOR 1 A.M. INTO REGISTER R1
```

(загрузить температуру в 1 час ночи в регистр R1), каждый процессор выполняет эту команду, используя свои собственные данные и свой регистр R1. Затем, когда программа вызывает команду

```
ADD THE TEMPERATURE FOR 2 A.M. TO REGISTER R1
```

(добавить температуру в 2 часа ночи в регистр R1), каждый процессор выполняет эту команду, используя свои собственные данные. В конце вычислений каждый процессор должен будет сосчитать среднюю температуру для каждого отдельного датчика.

Такая модель программирования имеет очень большое значение для аппаратного обеспечения. По существу, это значит, что каждый обрабатывающий элемент — это АЛУ и память, без схемы декодирования команд. Вместо этого один центральный блок вызывает команды и сообщает всем АЛУ, что делать дальше.

Альтернативная модель предполагает несколько потоков управления, каждый из которых содержит собственный счетчик команд, регистры и локальные переменные. Каждый поток управления выполняет свою собственную программу над своими данными, при этом он время от времени может взаимодействовать с другими потоками управления. Существует множество вариаций этой идеи, и в совокупности они формируют основную модель для параллельной обработки. По этой причине мы сосредоточимся на параллельной обработке с несколькими потоками контроля.

Степень распараллеливания процессов

Параллелизм управления можно вводить на разных уровнях. На самом низком уровне элементы параллелизма могут содержаться в отдельных машинных командах (например, в архитектуре IA-64). Программисты обычно не знают о существовании такого параллелизма — он управляется компилятором или аппаратным обеспечением.

На более высоком уровне мы приходим к **параллелизму на уровне блоков** (block-level parallelism), который позволяет программистам самим контролировать, какие высказывания будут выполняться последовательно, а какие — параллельно. Например, в языке Algol-68 выражение

```
begin Statement-1; Statement-2; Statement-3 end
```

использовалось для создания блока трех (в данном случае трех) произвольных высказываний, которые должны выполняться последовательно, а выражение

```
begin Statement-1, Statement-2, Statement-3 end
```

использовалось для параллельного выполнения тех же трех высказываний. С помощью правильного размещения точек с запятой, запятых, скобок и разграничителей **begin/end** можно было записать произвольную комбинацию команд для последовательного или параллельного выполнения.

Присутствует параллелизм на уровне более крупных структурных единиц, когда можно вызвать процедуру и не заставлять вызывающую программу ждать завершения этой процедуры. Это значит, что вызывающая программа и вызванная процедура будут работать параллельно. Если вызывающая программа находится в цикле, который вызывает процедуру при каждом прохождении и не ждет завершения этих процедур, то значит, большое число параллельных процедур запускается одновременно.

Другая форма параллелизма — создание или порождение для каждого процесса нескольких **потоков**, каждый из которых работает в пределах адресного пространства этого процесса. Каждый поток имеет свой счетчик команд, свои регистры и стек, но разделяет все остальное адресное пространство (а также все глобальные переменные) со всеми другими потоками. (В отличие от потоков разные процессы не разделяют общего адресного пространства.) Потоки работают независимо друг от друга, иногда на разных процессорах. В одних системах операционная система располагает информацией обо всех потоках и осуществляет планирование потоков; в других системах каждый пользовательский процесс сам выполняет планирование потоков и управляет потоками, а операционной системе об этом неизвестно.

Наконец, параллелизм на уровне еще более крупных структурных единиц — несколько независимых процессов, которые вместе работают над решением одной задачи. В отличие от потоков независимые процессы не разделяют общее адресное пространство, поэтому задача должна быть разделена на довольно большие куски, по одному на каждый процесс.

Вычислительные парадигмы

В большинстве параллельных программ, особенно в тех, которые содержат большое число потоков или независимых процессов, используется некоторая парадигма для структуризации их работы. Существует множество таких парадигм. В этом разделе мы упомянем только несколько самых популярных.

Первая парадигма — **SPMD (Single Program Multiple Data — одна программа, несколько потоков данных)**. Хотя система состоит из нескольких независимых процессов, они все выполняют одну и ту же программу, но над разными наборами данных. Только сейчас, в отличие от примера с температурой, все процессы выполняют одни и те же вычисления, но каждый в своем пространстве.

Вторая парадигма — **конвейер** с тремя процессами (рис. 8.11, *а*). Данные поступают в первый процесс, который трансформирует их и передает второму процессу для чтения и т. д. Если поток данных длинный (например, если это видеоизображение), все процессоры могут быть заняты одновременно. Так работают конвейеры в системе UNIX. Они могут работать как отдельные процессы параллельно в мультикомпьютере или мультипроцессоре.

Следующая парадигма — фазированное вычисление (рис. 8.11, *б*), когда работа разделяется на фазы, например, повторения цикла. Во время каждой фазы несколько процессов работают параллельно, но если один из процессов закончит

свою работу, он должен ждать до тех пор, пока все остальные процессы не завершат свою работу, и только после этого начинается следующая фаза. Четвертая парадигма — **«разделяй и властвуй»**, изображенная на рис. 8.11, *в*, в которой один процесс запускается, а затем порождает другие процессы, которым он может передать часть работы. Можно провести аналогию с генеральным подрядчиком, который получает приказ, затем поручает значительную часть работы каменщикам, электрикам, водопроводчикам, малярам и другим подчиненным. Каждый из них, в свою очередь, может поручить часть своей работы другим рабочим.

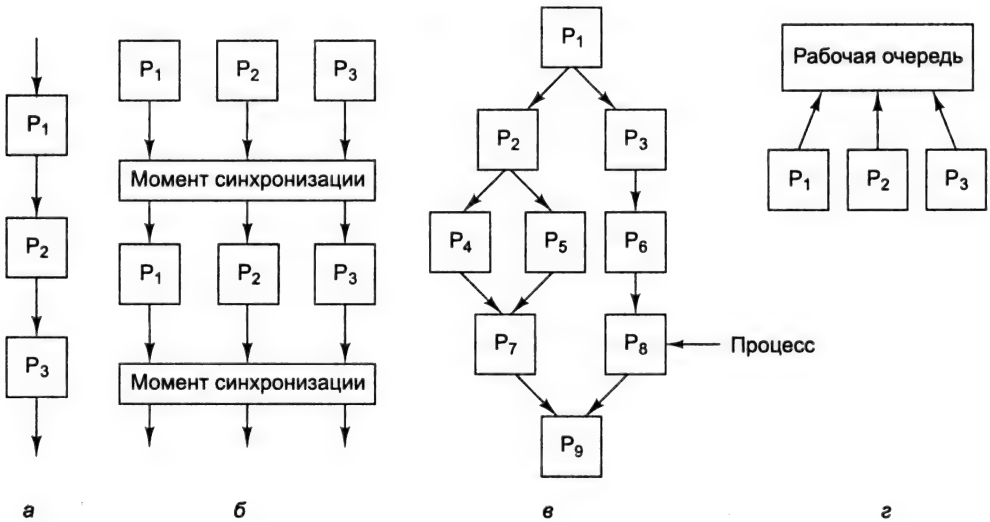


Рис. 8.11. Вычислительные парадигмы: конвейер (*а*); фазированное вычисление (*б*); «разделяй и властвуй» (*в*); replicated worker (*г*)

Последний пример — парадигма replicated worker (см. рис. 8.11, *г*). Здесь существует центральная очередь, и рабочие процессы получают задачи из этой очереди и выполняют их. Если задача порождает новые задачи, они добавляются к центральной очереди. Каждый раз, когда рабочий процесс завершает выполнение текущей задачи, он получает из очереди следующую задачу.

Методы коммуникации

Если программа разделена на части, скажем, на процессы, которые работают параллельно, эти части (процессы) должны каким-то образом взаимодействовать друг с другом. Такое взаимодействие можно осуществить одним из двух способов: с помощью общих переменных и с помощью передачи сообщений. В первом случае все процессы имеют доступ к общей логической памяти и взаимодействуют, считывая и записывая информацию в эту память. Например, один процесс может установить переменную, а другой процесс может прочитать ее.

В мультипроцессоре переменные могут разделяться между несколькими процессами с помощью отображения одной и той же страницы в адресное пространство каждого процесса. Затем общие переменные можно считывать и записывать с помощью обычных машинных команд LOAD и STORE. Даже в мультикомпьютере

без разделенной физической памяти возможно логическое разделение переменных, но сделать это несколько сложнее. Как мы видели выше, в системе Linda существует общее пространство кортежей, даже на мультикомпьютере, а в системе Orca поддерживаются разделенные объекты, общие с другими машинами. Существует и возможность разделения одного адресного пространства на мультикомпьютере, а также разбиение на страницы в сети. Процессы могут взаимодействовать между собой через общие переменные как в мультипроцессорах, так и в мультикомпьютерах.

Альтернативный подход — взаимодействие через передачу сообщений. В данной модели используются примитивы `send` и `receive`. Один процесс выполняет примитив `send`, называя другой процесс в качестве пункта назначения. Как только второй процесс совершает `receive`, сообщение копируется в адресное пространство получателя.

Существует множество вариантов передачи сообщений, но все они сводятся к применению двух примитивов `send` и `receive`. Они обычно реализуются как системные вызовы. Более подробно этот процесс мы рассмотрим ниже в этой главе.

Следующий вопрос при передаче сообщений — количество получателей. Самый простой случай — один отправитель и один получатель (**двухточечная передача сообщений**). Однако в некоторых случаях требуется отправить сообщение всем процессам (**широковещание**) или определенному набору процессов (**мультивещание**).

Отметим, что в мультипроцессоре передачу сообщений легко реализовать путем простого копирования от отправителя к получателю. Таким образом, возможности физической памяти совместного использования (мультипроцессор/мультикомпьютер) и логической памяти совместного использования (взаимодействие через общие переменные/передача сообщений) никак не связаны между собой. Все четыре комбинации имеют смысл и могут быть реализованы. Они приведены в табл. 8.1.

Таблица 8.1. Комбинации совместного использования физической и логической памяти

Физическая память (аппаратное обеспечение)	Логическая память (программное обеспечение)	Примеры
Мультипроцессор	Разделяемые переменные	Обработка изображения (см. рис. 8.1).
Мультипроцессор	Передача сообщений	Передача сообщений с использованием буферов памяти
Мультикомпьютер	Разделяемые переменные	DSM, Linda, Orca и т. д. на SP/2 или сети персонального компьютера
Мультикомпьютер	Передача сообщений	PVM или MPI на SP/2 или сети персонального компьютера

Базисные элементы синхронизации

Параллельные процессы должны не только взаимодействовать, но и синхронизировать свои действия. Если процессы используют общие переменные, нужно быть уверенным, что пока один процесс записывает что-либо в общую структуру данных, никакой другой процесс не считывает эту структуру. Другими словами, требуется некоторая форма **взаимного исключения**, чтобы несколько процессов не могли использовать одни и те же данные одновременно.

Существуют различные базисные элементы, которые можно использовать для взаимного исключения. Это семафоры, блокировки, мьютексы и критические секции. Все они позволяют процессу использовать какой-то ресурс (общую переменную, устройство ввода-вывода и т. п.), и при этом никакие другие процессы доступа к этому ресурсу не имеют. Если получено разрешение на доступ, процесс может использовать этот ресурс. Если второй процесс запрашивает разрешение, а первый все еще использует этот ресурс, доступ будет запрещен до тех пор, пока первый процесс не освободит ресурс.

Во многих параллельных программах существует и такой вид примитивов (базисных элементов), которые блокируют все процессы до тех пор, пока определенная фаза работы не завершится (см. рис. 8.11, *б*). Наиболее распространенным примитивом подобного рода является **барьер**. Когда процесс встречает барьер, он блокируется до тех пор, пока все процессы не наткнутся на барьер. Когда последний процесс встречает барьер, все процессы одновременно освобождаются и продолжают работу.

Классификация компьютеров параллельного действия

Многое можно сказать о программном обеспечении для параллельных компьютеров, но сейчас мы должны вернуться к основной теме данной главы — архитектуре компьютеров параллельного действия. Было предложено и построено множество различных видов параллельных компьютеров, поэтому хотелось бы узнать, можно ли их как-либо категоризировать. Это пытались сделать многие исследователи [39, 43, 148]. К сожалению, хорошей классификации компьютеров параллельного действия до сих пор не существует. Чаще всего используется классификация Флинна (Flynn), но даже она является очень грубым приближением. Классификация приведена в табл. 8.2.

Таблица 8.2. Классификация компьютеров параллельного действия, разработанная Флинном

Потоки команд	Потоки данных	Названия	Примеры
1	1	SISD	Классическая машина фон Неймана
1	Много	SIMD	Векторный суперкомпьютер, массивно-параллельный процессор
Много	1	MISD	Не существует
Много	Много	MIMD	Мультипроцессор, мультикомпьютер

В основе классификации лежат два понятия: потоки команд и потоки данных. Поток команд соответствует счетчику команд. Система с n процессорами имеет n счетчиков команд и, следовательно, n потоков команд.

Поток данных состоит из набора операндов. В примере с вычислением температуры, приведенном выше, было несколько потоков данных, один для каждого датчика.

Потоки команд и данных в какой-то степени независимы, поэтому существует 4 комбинации (см. табл. 8.2). SISD (Single Instruction stream Single Data stream —

один поток команд, один поток данных) — это классический последовательный компьютер фон Неймана. Он содержит один поток команд и один поток данных и может выполнять только одно действие одномоментно. Машины SIMD (Single Instruction stream Multiple Data stream — один поток команд, несколько потоков данных) содержат один блок управления, выдающий по одной команде, но при этом есть несколько АЛУ, которые могут обрабатывать несколько наборов данных одновременно. ILLIAC IV (см. рис. 2.6) — прототип машин SIMD. Существуют и современные машины SIMD. Они применяются для научных вычислений.

Машины MISD (Multiple Instruction stream Single Data stream — несколько потоков команд, один поток данных) — несколько странная категория. Здесь несколько команд оперируют одним набором данных. Трудно сказать, существуют ли такие машины. Однако некоторые считают машинами MISD машины с конвейерами.

Последняя категория — машины MIMD (Multiple Instruction stream Multiple Data stream — несколько потоков команд, несколько потоков данных). Здесь несколько независимых процессоров работают как часть большой системы. В эту категорию попадает большинство параллельных процессоров. И мультипроцессоры, и мультикомпьютеры — это машины MIMD.

Мы расширили классификацию Флинна (схема на рис. 8.12). Машины SIMD распались на две подгруппы. В первую подгруппу попадают многочисленные суперкомпьютеры и другие машины, которые оперируют векторами, выполняя одну и ту же операцию над каждым элементом вектора. Во вторую подгруппу попадают машины типа ILLIAC IV, в которых главный блок управления посылает команды нескольким независимым АЛУ.

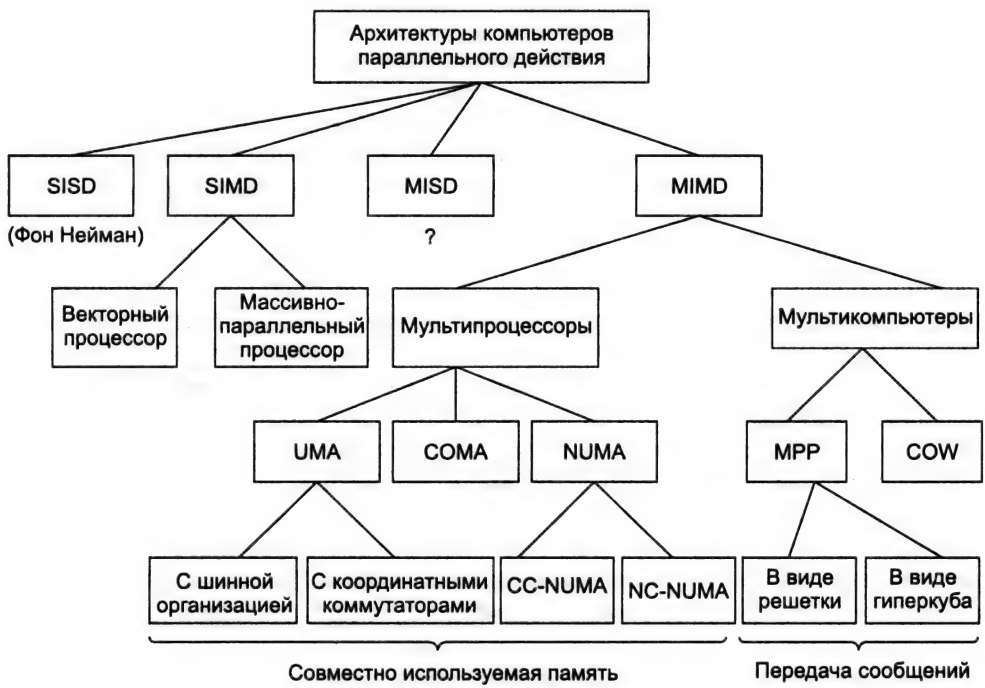


Рис. 8.12. Классификация компьютеров параллельного действия

В нашей классификации категория MIMD распалась на мультипроцессоры (машины с памятью совместного использования) и мультикомпьютеры (машины с передачей сообщений). Существует три типа мультипроцессоров. Они отличаются друг от друга по способу реализации памяти совместного использования. Они называются **UMA (Uniform Memory Access — архитектура с однородным доступом к памяти)**, **NUMA (NonUniform Memory Access — архитектура с неоднородным доступом к памяти)** и **COMA (Cache Only Memory Access — архитектура с доступом только к кэш-памяти)**. В машинах UMA каждый процессор имеет одно и то же время доступа к любому модулю памяти. Иными словами, каждое слово памяти можно считать с той же скоростью, что и любое другое слово памяти. Если это технически невозможно, самые быстрые обращения замедляются, чтобы соответствовать самым медленным, поэтому программисты не увидят никакой разницы. Это и значит «однородный». Такая однородность делает производительность предсказуемой, а этот фактор очень важен для написания эффективной программы.

Мультипроцессор NUMA, напротив, не обладает этим свойством. Обычно есть такой модуль памяти, который расположен близко к каждому процессору, и доступ к этому модулю памяти происходит гораздо быстрее, чем к другим. С точки зрения производительности очень важно, куда помещаются программа и данные. Машины COMA тоже с неоднородным доступом, но по другой причине. Подробнее каждый из этих трех типов мы рассмотрим позднее, когда будем изучать соответствующие подкатегории.

Во вторую подкатегорию машин MIMD попадают мультикомпьютеры, которые в отличие от мультипроцессоров не имеют памяти совместного использования на архитектурном уровне. Другими словами, операционная система в процессоре мультикомпьютера не может получить доступ к памяти, относящейся к другому процессору, просто путем выполнения команды `LOAD`. Ему приходится отправлять сообщение и ждать ответа. Именно способность операционной системы считывать слово из отдаленного модуля памяти с помощью команды `LOAD` отличает мультипроцессоры от мультикомпьютеров. Как мы уже говорили, даже в мультикомпьютере пользовательские программы могут обращаться к другим модулям памяти с помощью команд `LOAD` и `STORE`, но эту иллюзию создает операционная система, а не аппаратное обеспечение. Разница незначительна, но очень важна. Так как мультикомпьютеры не имеют прямого доступа к отдаленным модулям памяти, они иногда называются машинами **NORMA (NO Remote Memory Access — без доступа к отдаленным модулям памяти)**.

Мультикомпьютеры можно разделить на две категории. Первая категория содержит процессоры **MPP (Massively Parallel Processors — процессоры с массовым параллелизмом)** — дорогостоящие суперкомпьютеры, которые состоят из большого количества процессоров, связанных высокоскоростной коммуникационной сетью. В качестве примеров можно назвать Cray T3E и IBM SP/2.

Вторая категория мультикомпьютеров включает рабочие станции, которые связываются с помощью уже имеющейся технологии соединения. Эти примитивные машины называются **NOW (Network of Workstations — сеть рабочих станций)** и **COW (Cluster of Workstattions — кластер рабочих станций)**.

В следующих разделах мы подробно рассмотрим машины SIMD, мультипроцессоры MIMD и мультикомпьютеры MIMD. Цель — показать, что собой представляет каждый из этих типов, что собой представляют подкатегории и каковы ключевые принципы разработки. В качестве иллюстраций мы рассмотрим несколько конкретных примеров.

Компьютеры SIMD

Компьютеры SIMD (Single Instruction Stream Multiple Data Stream — один поток команд, несколько потоков данных) используются для решения научных и технических задач с векторами и массивами. Такая машина содержит один блок управления, который выполняет команды по одной, но каждая команда оперирует несколькими элементами данных. Два основных типа компьютеров SIMD — это массивно-параллельные процессоры (array processors) и векторные процессоры (vector processors). Рассмотрим каждый из этих типов по отдельности.

Массивно-параллельные процессоры

Идея массивно-параллельных процессоров была впервые предложена более 40 лет назад [151]. Однако прошло еще около 10 лет, прежде чем такой процессор (ILLIAC IV) был построен для NASA. С тех пор другие компании создали несколько коммерческих массивно-параллельных процессоров, в том числе CM-2 и Maspar MP-2, но ни один из них не пользовался популярностью на компьютерном рынке.

В **массивно-параллельном процессоре** содержится один блок управления, который передает сигналы, чтобы запустить несколько обрабатывающих элементов, как показано на рис. 2.6. Каждый обрабатывающий элемент состоит из процессора или усовершенствованного АЛУ и, как правило, локальной памяти.

Хотя все массивно-параллельные процессоры соответствуют этой общей модели, они могут отличаться друг от друга в некоторых моментах. Первый вопрос — это структура обрабатывающего элемента. Она может быть различной — от чрезвычайно простой до чрезвычайно сложной. Самые простые обрабатывающие элементы — 1-битные АЛУ (как в CM-2). В такой машине каждый АЛУ получает два 1-битных операнда из своей локальной памяти плюс бит из слова состояния программы (например, бит переноса). Результат операции — 1 бит данных и несколько флаговых битов. Чтобы совершить сложение двух целых 32-битных чисел, блоку управления нужно транслировать команду 1-битного сложения 32 раза. Если на одну команду затрачивается 600 нс, то для сложения целых чисел потребуется 19,2 мкс, то есть получается медленнее, чем в первоначальной IBM PC. Но при наличии 65 536 обрабатывающих элементов можно получить более трех миллиардов сложений в секунду при времени сложения 300 пикосекунд.

Обрабатывающим элементом может быть 8-битное АЛУ, 32-битное АЛУ или более мощное устройство, способное выполнять операции с плавающей точкой. В какой-то степени выбор типа обрабатывающего элемента зависит от типа целей машины. Операции с плавающей точкой могут потребоваться для сложных мате-

матических расчетов (хотя при этом существенно сократится число обрабатывающих элементов), но для информационного поиска они не нужны.

Второй вопрос — как связываются обрабатывающие элементы друг с другом. Здесь применимы практически все топологии, приведенные на рис. 8.4. Чаще всего используются прямоугольные решетки, поскольку они подходят для задач с матрицами и отображениями и хорошо применимы к большим размерам, так как с добавлением новых процессоров автоматически увеличивается пропускная способность.

Третий вопрос — какую локальную автономию имеют обрабатывающие элементы. Блок управления сообщает, какую команду нужно выполнить, но во многих массивно-параллельных процессорах каждый обрабатывающий элемент может выбирать на основе некоторых локальных данных (например, на основе битов кода условия), выполнять ему эту команду или нет. Эта особенность придает процессору значительную гибкость.

Векторные процессоры

Второй тип машины SIMD — **векторный процессор**. Он более популярен на рынке. Машины, разработанные Сеймуром Креем (Seymour Cray) для Cray Research (сейчас это часть Silicon Graphics), — Cray-1 в 1976 году, а затем C90 и T90 доминировали в научной сфере на протяжении десятилетий. В этом разделе мы рассмотрим основные принципы, которые используются при создании таких высокопроизводительных компьютеров.

Типичное приложение для быстрой переработки больших объемов цифровых данных полно таких выражений, как:

```
for(i=0; i<n; i++) a[i]=b[i]+c[i];
```

где a, b и c — это **векторы**[1] (массивы чисел), обычно с плавающей точкой. Цикл приказывает компьютеру сложить i-е элементы векторов b и c, и сохранить результат в i-м элементе массива a. Программа определяет, что элементы должны складываться последовательно, но обычно порядок не играет никакой роли.

На рис. 8.13 изображено векторное АЛУ. Такая машина на входе получает два n-элементных вектора и обрабатывает соответствующие элементы параллельно, используя векторное АЛУ, которое может оперировать n элементами одновременно. В результате получается вектор. Входные и выходные векторы могут сохраняться в памяти или в специальных векторных регистрах.

Векторные компьютеры применяются и для скалярных (невекторных) операций, а также для смешанных векторно-скалярных операций. Основные типы векторных операций приведены в табл. 8.3. Первая из них, f_1, выполняет ту или иную операцию (например, квадратный корень или косинус) над каждым элементом одного вектора. Вторая, f_2, на входе получает вектор, а на выходе выдает скалярное значение. Типичный пример — суммирование всех элементов. Третья, f_3, выполняет бинарную операцию над двумя векторами, например сложение соответствую-

[1] Строго говоря, использование термина «вектор» в данном контексте неправомочно, хотя уже много лет говорят именно «вектор». Дело в том, что вектор, в отличие от одномерной матрицы, имеет метрику, тогда как одномерный массив представляет собой просто определенным образом упорядоченный набор значений, характеризующих некоторый объект. — *Примеч. научн. ред.*

щих элементов. Наконец, четвертая, f_4, соединяет скалярный операнд с векторным. Типичный пример — умножение каждого элемента вектора на константу. Иногда быстрее переделать скалярный операнд в вектор, каждое значение которого равно скалярному операнду, а затем выполнить операцию над двумя векторами.

Все обычные операции с векторами могут производиться с использованием этих четырех форм. Например, чтобы получить скалярное произведение двух векторов, нужно сначала перемножить соответствующие элементы векторов (f_3), а затем сложить полученные результаты (f_2).

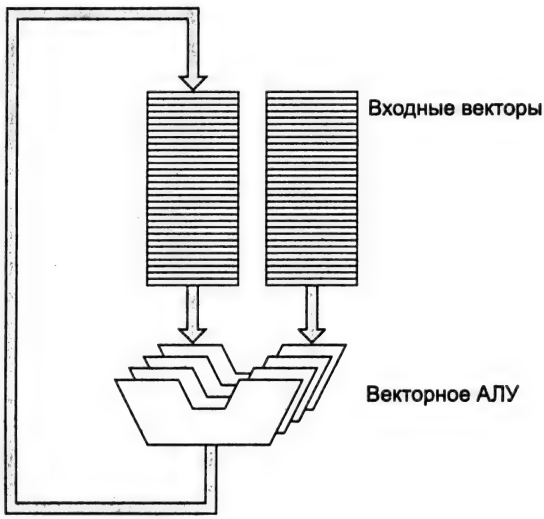


Рис. 8.13. Векторное АЛУ

Таблица 8.3. Комбинации векторных и скалярных операций

Операция	Примеры
$A_i=f_1(B_i)$	f_1 – косинус, квадратный корень
Скаляр=$f_2(A)$	f_2 — сумма, минимум
$A_i=f_3(B_i, C_i)$	f_3 — сложение, вычитание
$A_i=f_4$(скаляр, B_i)	f_4 – умножение B_i на константу

На практике суперкомпьютеры редко строятся по схеме, изображенной на рис. 8.13. Причина не техническая — такую машину вполне можно построить, а экономическая. Создание компьютера с 64 высокоскоростными АЛУ слишком дорого обойдется.

Обычно векторные процессоры сочетаются с конвейерами. Операции с плавающей точкой достаточно сложны. Они требуют выполнения нескольких шагов, а для выполнения любой многошаговой операции лучше использовать конвейер. Если вы не знакомы с арифметикой с плавающей точкой, обратитесь к приложению Б.

Рассмотрим табл. 8.4. В данном примере нормализованное число имеет мантиссу больше 1, но меньше 10 или равную 1. Здесь требуется вычесть $9{,}212\times10^{11}$ из $1{,}082\times10^{12}$.

Таблица 8.4. Вычитание чисел с плавающей точкой

Номер шага	Название шага	Значения
1	Вызов операндов	$1{,}082\times10^{12}$–$9{,}212\times10^{11}$
2	Выравнивание экспоненты	$1{,}082\times10^{12}$–$0{,}9212\times10^{12}$
3	Вычитание	$0{,}1608\times10^{12}$
4	Нормализация результата	$1{,}608\times10^{11}$

Чтобы из одного числа с плавающей точкой вычесть другое число с плавающей точкой, сначала нужно подогнать их таким образом, чтобы их экспоненты имели одно и то же значение. В нашем примере мы можем либо преобразовать уменьшаемое (число, из которого производится вычитание) в $10{,}82\times10^{11}$, либо преобразовать вычитаемое (число, которое вычитается) в $0{,}9212\times10^{12}$. При любом преобразовании мы рискуем. Увеличение экспоненты может привести к антипереполнению (исчезновению значащих разрядов) мантиссы, а уменьшение экспоненты может вызвать переполнение мантиссы. Антипереполнение менее опасно, поскольку число с антипереполнением можно округлить нулем. Поэтому мы выбираем первый путь. Приведя обе экспоненты к 12, мы получаем значения, которые показаны в табл. 8.4 на шаге 2. Затем мы выполняем вычитание, а потом нормализуем результат.

Конвейеризацию можно применять к циклу `for`, приведенному в начале раздела. В табл. 8.5 показана работа конвейеризированного сумматора с плавающей точкой. В каждом цикле на первой стадии вызывается пара операндов. На второй стадии меньшая экспонента подгоняется таким образом, чтобы соответствовать большей. На третьей стадии выполняется операция, а на четвертой стадии нормализуется результат. Таким образом, в каждом цикле из конвейера выходит один результат.

Таблица 8.5. Работа конвейеризированного сумматора с плавающей точкой

	Цикл						
Шаг	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Вызов операндов	B_1, C_1	B_2, C_2	B_3, C_3	B_4, C_4	B_5, C_5	B_6, C_6	B_7, C_7
Выравнивание экспоненты		B_1, C_1	B_2, C_2	B_3, C_3	B_4, C_4	B_5, C_5	B_6, C_6
Вычитание			B_1+C_1	B_2+C_2	B_3+C_3	B_4+C_4	B_5+C_5
Нормализация результата				B_1+C_1	B_2+C_2	B_3+C_3	B_4+C_4

Существенное различие между использованием конвейера для операций над векторами и использованием его для выполнения обычных команд — отсутствие скачков при работе с векторами. Каждый цикл используется полностью, и никаких пустых циклов нет.

Векторный суперкомпьютер Cray-1

Суперкомпьютеры обычно содержат несколько АЛУ, каждое из которых предназначено для выполнения определенной операции, и все они могут работать параллельно. В качестве примера рассмотрим суперкомпьютер Cray-1. Он больше не используется, но зато имеет простую архитектуру типа RISC, поэтому его очень удобно применять в учебных целях, и к тому же его архитектуру можно встретить во многих современных векторных суперкомпьютерах.

Машина Cray-1 регистровая (что типично для машин типа RISC), большинство команд 16-битные, состоят из 7-битного кода операции и трех 3-битных номеров регистров для трех операндов. Имеется пять типов регистров (рис. 8.14). Восемь 24-битных регистров A используются для обращения к памяти. 64 24-битных регистра B используются для хранения регистров A, когда они не нужны, чтобы не записывать их обратно в память. Восемь 64-битных регистров S предназначены для хранения скалярных величин (целых чисел и чисел с плавающей точкой). Значения этих регистров можно использовать в качестве операндов как для операций с целыми числами, так и для операций над числами с плавающей точкой. 64 64-битных регистра T — это регистры для хранения регистров S, опять-таки для сокращения количество операций LOAD и STORE.

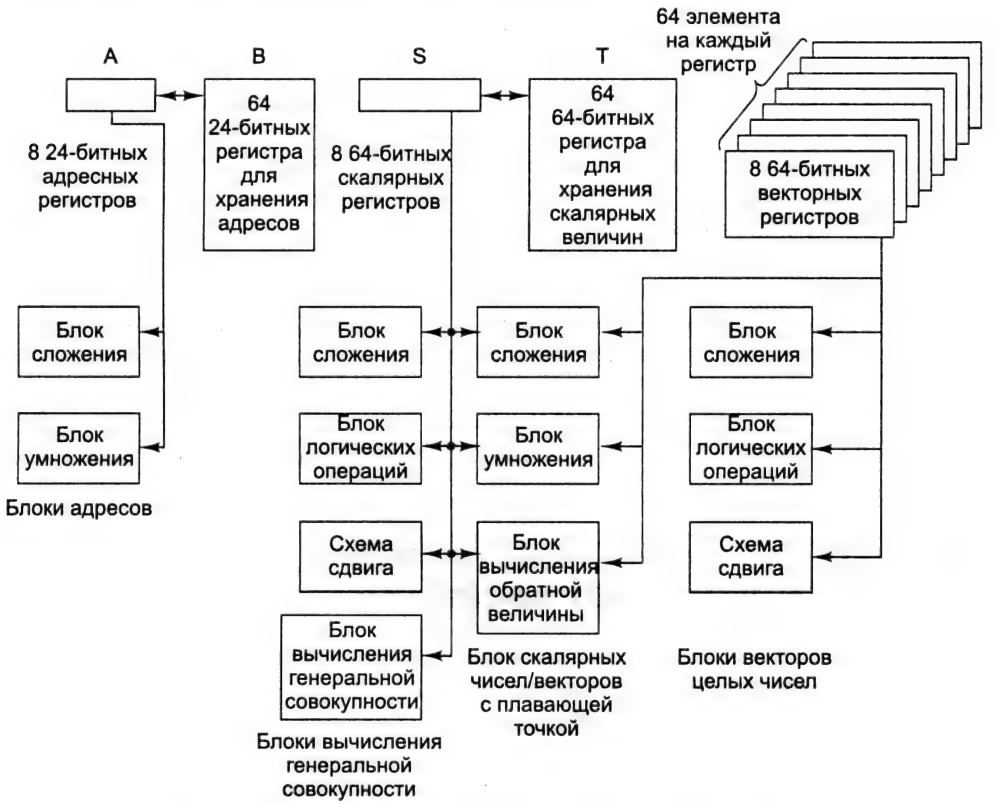


Рис. 8.14. Регистры и функциональные блоки машины Cray-1

Самый интересный набор регистров — это группа из восьми векторных регистров. Каждый такой регистр может содержать 64-элементный вектор с плавающей точкой. В одной 16-битной команде можно сложить, вычесть или умножить два вектора. Операция деления невозможна, но можно вычислить обратную величину. Векторные регистры могут загружаться из памяти, сохраняться в память, но такие перемещения выполнять очень невыгодно, поэтому их следует свести к минимуму. Во всех векторных операциях используются регистровые операнды.

Не всегда в суперкомпьютерах операнды должны находиться в регистрах. Например, машина CDC Cyber 205 выполняла операции над векторами в памяти. Такой подход позволял работать с векторами произвольной длины, но это сильно снижало скорость работы машины.

Cray-1 содержит 12 различных функциональных блоков (см. рис. 8.14). Два из них предназначены для арифметических действий с 24-битными адресами. Четыре нужны для операций с 64-битными целыми числами. Cray-1 не имеет блока для целочисленного умножения (хотя есть блок для перемножения чисел с плавающей точкой). Оставшиеся шесть блоков работают над векторами. Все они конвейеризированы. Блоки сложения, умножения и вычисления обратной величины работают как над скалярными числами с плавающей точкой, так и над векторами.

Как и многие другие векторные компьютеры, Cray-1 допускает операции сцепления. Например, чтобы вычислить выражение

```
R1=R1*R2+R3
```

где R1, R2 и R3 — векторные регистры, машина выполнит векторное умножение элемент за элементом, сохранит результат где-нибудь, а затем выполнит векторное сложение. При сцеплении, как только первые элементы перемножены, произведение сразу направляется в сумматор вместе с первым элементом регистра R3. Сохранения промежуточного результата не требуется. Такая технология значительно улучшает производительность.

Интересно рассмотреть абсолютную производительность Cray-1. Тактовый генератор работает с частотой 80 МГц, а объем основной памяти составляет 8 Мбайт. В то время (в середине — конце 70-х) это был самый мощный компьютер в мире. Сегодня вряд ли кто-нибудь сможет купить компьютер с таким медленным тактовым генератором и такой маленькой памятью — их уже никто не производит. Это наблюдение показывает, как быстро развивается компьютерная промышленность.

Мультипроцессоры с памятью совместного использования

Как показано на рис. 8.12, системы MIMD можно разделить на мультипроцессоры и мультикомпьютеры. В этом разделе мы рассмотрим мультипроцессоры, а в следующем — мультикомпьютеры. Мультипроцессор — это компьютерная система, которая содержит несколько процессоров и одно адресное пространство, видимое для всех процессоров. Он запускает одну копию операционной системы с одним набором таблиц, в том числе таблицами, которые следят, какие страницы памяти заняты, а какие свободны. Когда процесс блокируется, его процессор сохраняет свое состояние в таблицах операционной системы, а затем просматривает эти таблицы для нахождения другого процесса, который нужно запустить. Именно наличие одного отображения и отличает мультипроцессор от мультикомпьютера.

Мультипроцессор, как и все компьютеры, должен содержать устройства ввода-вывода (диски, сетевые адаптеры и т. п.). В одних мультипроцессорных системах только определенные процессоры имеют доступ к устройствам ввода-вывода и, следовательно, имеют специальную функцию ввода-вывода. В других мультипро-

цессорных системах каждый процессор имеет доступ к любому устройству ввода-вывода. Если все процессоры имеют равный доступ ко всем модулям памяти и всем устройствам ввода-вывода и каждый процессор взаимозаменим с другими процессорами, то такая система называется **SMP (Symmetric MultiProcessor — симметричный мультипроцессор)**. Ниже мы будем говорить именно о таком типе систем.

Семантика памяти

Несмотря на то, что во всех мультипроцессорах процессорам предоставляется отображение общего разделенного адресного пространства, часто наряду с этим имеется множество модулей памяти, каждый из которых содержит какую-либо часть физической памяти. Процессоры и модули памяти соединяются сложной коммуникационной сетью (мы это обсуждали в разделе «Сети межсоединений»). Несколько процессоров могут пытаться считать слово из памяти, а в это же время несколько других процессоров пытаются записать то же самое слово, и некоторые сообщения могут быть доставлены не в том порядке, в каком они были отправлены. Добавим к этой проблеме существование многочисленных копий некоторых блоков памяти (например, в кэш-памяти), и в результате мы придем к хаосу, если не принять определенные меры. В этом разделе мы увидим, что в действительности представляет собой память совместного использования, и посмотрим, как блоки памяти могут правильно реагировать при таких обстоятельствах.

Семантику памяти можно рассматривать как контракт между программным обеспечением и аппаратным обеспечением памяти [3]. Если программное обеспечение соглашается следовать определенным правилам, то память соглашается выдавать определенные результаты. Основная проблема здесь — каковы должны быть правила. Эти правила называются **моделями согласованности**. Было предложено и разработано множество таких правил.

Предположим, что процессор 0 записывает значение 1 в какое-то слово памяти, а немного позже процессор 1 записывает значение 2 в то же самое слово. Процессор 2 считывает это слово и получает значение 1. Должен ли владелец компьютера обратиться после этого в мастерскую? Это зависит от того, что обещано в контракте.

Строгая согласованность

Самая простая модель — **строгая согласованность**. В такой модели при любом считывании из адреса x всегда возвращается значение самой последней записи в x. Программистам очень нравится эта модель, но ее можно реализовать на практике только следующим образом: должен быть один модуль памяти, который просто обслуживает все запросы по мере поступления (первым поступил — первым обработан), без кэш-памяти и без дублирования данных. К несчастью, это очень сильно замедляет работу памяти.

Согласованность по последовательности

Следующая модель называется **согласованностью по последовательности** [79]. Здесь при наличии нескольких запросов на чтение и запись аппаратное обеспечение определяет порядок всех запросов, но все процессоры наблюдают одну и ту же последовательность запросов.

Рассмотрим один пример. Предположим, что процессор 1 записывает значение 100 в слово x, а через 1 нс процессор 2 записывает значение 200 в слово x. А теперь предположим, что через 1 нс после начала второй записи (процесс записи может быть еще не закончен) два других процессора, 3 и 4, считывают слово x по два раза (рис. 8.15). Возможный порядок шести событий представлен в листингах, относящихся к этому рисунку. В первом из них процессор 3 получает значения (200, 200), и процессор 4 получает значения (200, 200). Во втором они получают (100, 200) и (200, 200) соответственно. В третьем они получают (100, 100) и (200, 100) соответственно. Все эти варианты допустимы, как и некоторые другие, которые мы здесь не показали.



Рис. 8.15. Два процессора записывают, а другие два процессора считывают одно и то же слово из общей памяти

В листингах 8.1—8.3 представлены возможные варианты двух записей и четырех чтений.

Листинг 8.1. К рис. 8.15 (а)

```
W100
W200
R3=200
R3=200
R4=200
R4=200
```

Листинг 8.2. К рис. 8.15 (б)

```
W100
R3=100
W200
R4=200
R3=200
R4=200
```

Листинг 8.3. К рис. 8.15 (в)

```
W200
R4=200
W100
R3=100
R4=100
R3=100
```

Согласованная по последовательности память никогда не позволит процессору 3 получить значение (100, 200), если процессор 4 получил (200, 100). Если бы это произошло, с точки зрения процессора 3 это бы означало, что запись значения 100 процессором 1 завершилась раньше записи значения 200, которую осуществляет процессор 2. Это вполне возможно. Но с точки зрения процессора 4 это также значит, что запись процессором 2 числа 200 завершилась до записи процессором 1 числа 100. Сам по себе такой результат тоже возможен, но он противоречит первому результату. Согласованность по последовательности гарантирует единый глобальный порядок записей, который виден всем процессорам. Если процессор 3 видит, что первым было записано значение 100, то процессор 4 должен видеть тот же порядок.

Согласованность по последовательности очень полезна. Если несколько событий совершаются одновременно, существует определенный порядок, в котором эти события происходят (порядок может определяться случайно), но все процессоры наблюдают тот же самый порядок. Ниже мы рассмотрим модели согласованности, которые не гарантируют такого порядка.

Процессорная согласованность

Процессорная согласованность [48] — более проигрышная модель, но зато ее легче реализовать на больших мультипроцессорах. Она имеет два свойства:

1. Все процессоры воспринимают записи любого процессора в том порядке, в котором они начинаются.
2. Все процессоры видят записи в любое слово памяти в том же порядке, в котором они происходят.

Эти два пункта очень важны. В первом пункте говорится, что если процессор 1 начинает запись значений 1A, 1B и 1C в какое-либо место в памяти именно в таком порядке, то все другие процессоры видят эти записи в том же порядке. Иными словами, никогда не произойдет такого, чтобы какой-либо процессор сначала увидел значение 1B, а затем значение 1A. Второй пункт нужен, чтобы каждое слово в памяти имело определенное недвусмысленное значение после того, как процессор совершил несколько записей в это слово, а затем остановился. Все должны воспринимать последнее значение.

Даже при таких ограничениях у разработчика есть много возможностей. Посмотрим, что произойдет, если процессор 2 начинает записи 2A, 2B и 2C одновременно с тремя записями процессора 1. Другие процессоры, которые заняты считыванием слов из памяти, увидят какую-либо последовательность из шести записей, например, 1A, 1B, 2A, 2B, 1C, 2C или 2A, 1A, 2B, 2C, 1B, 1C и т. п. При процессорной согласованности не гарантируется, что каждый процессор видит один и тот же порядок (в отличие от согласованности по последовательности). Вполне может быть так, что одни процессоры воспринимают первый порядок из указанных выше, другие — второй порядок, а третьи — иной третий порядок. Единственное, что гарантируется абсолютно точно, — ни один процессор не увидит последовательность, в которой сначала идет 1B, а затем 1A.

Слабая согласованность

В следующей модели, **слабой согласованности**, записи, произведенные одним процессором, воспринимаются по порядку [33]. Один процессор может увидеть 1A до 1B, а другой — 1A после 1B. Чтобы внести порядок в этот хаос, в памяти содержатся переменные синхронизации либо операция синхронизации. Когда выполняется синхронизация, все незаконченные записи завершаются и ни одна новая запись не может начаться, пока не будут завершены все старые записи и не будет произведена синхронизация. Синхронизация приводит память в стабильное состояние, когда не остается никаких незавершенных операций. Сами операции синхронизации согласованы по последовательности, то есть если они вызываются несколькими процессорами, выбирается какой-то определенный порядок, причем все процессоры воспринимают один и тот же порядок.

При слабой согласованности время разделяется на последовательные периоды, разграниченные моментами синхронизации (рис. 8.16). Никакого определенного порядка для 1A и 1B не гарантируется, и разные процессоры могут воспринимать эти две записи в разном порядке, то есть один процессор может сначала видеть 1A, а затем 1B, а другой процессор может сначала видеть 1B, а затем 1A. Такая ситуация допустима. Однако все процессоры видят сначала 1B, а затем 1C, поскольку первая операция синхронизации требует, чтобы сначала завершились записи 1A, 1B и 2A, и только после этого начались записи 1C, 2B, 3A или 3B. Таким образом, с помощью операций синхронизации программное обеспечение может вносить порядок в последовательность событий, хотя это занимает некоторое время.

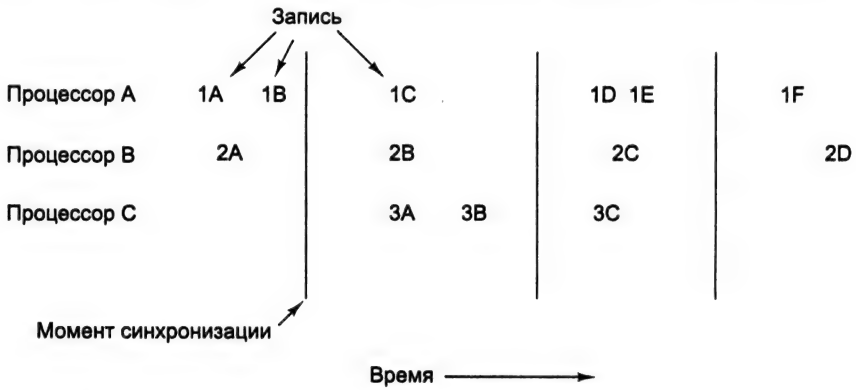


Рис. 8.16. Слабо согласованная память использует операции синхронизации, которые делят время на последовательные периоды

Свободная согласованность

Слабая согласованность — не очень эффективный метод, поскольку он требует завершения всех операций памяти и задерживает выполнение новых операций до тех пор, пока старые не будут завершены. При **свободной согласованности** дела обстоят гораздо лучше, поскольку здесь используется нечто похожее на критические секции программы [46]. Идея состоит в следующем. Если процесс выходит за пределы критической области, это не значит, что все записи должны немедленно

завершиться. Требуется только, чтобы все записи были завершены до того, как любой процесс снова войдет в эту критическую область.

В этой модели операция синхронизации разделяется на две разные операции. Чтобы считать или записать общую переменную, процессор (то есть его программное обеспечение) сначала должен выполнить операцию `acquire` над переменной синхронизации, чтобы получить монопольный доступ к общим разделяемым данным. Затем процессор может использовать эти данные по своему усмотрению (считывать или записывать их). Потом процессор выполняет операцию `release` над переменной синхронизации, чтобы показать, что он завершил работу. Операция `release` не требует завершения незаконченных записей, но сама она не может быть завершена, пока не закончатся все ранее начатые записи. Более того, новые операции памяти могут начинаться сразу же.

Когда начинается следующая операция `acquire`, производится проверка, все ли предыдущие операции `release` завершены. Если нет, то операция `acquire` задерживается до тех пор, пока они все не будут сделаны (а все записи должны быть завершены перед тем, как завершатся все операции `release`). Таким образом, если следующая операция `acquire` появляется через достаточно длительный промежуток времени после последней операции `release`, ей не нужно ждать, и она может войти в критическую область без задержки. Если операция `acquire` появляется через небольшой промежуток времени после операции `release`, эта операция `acquire` (и все команды, которые следуют за ней) будет задержана до завершения всех операций `release`. Это гарантирует, что все переменные в критической области будут обновлены. Такая схема немного сложнее, чем слабая согласованность, но она имеет существенное преимущество: здесь не нужно задерживать выполнение команд так часто, как в слабой согласованности.

Архитектуры UMA SMP с шинной организацией

В основе самых простых мультипроцессоров лежит одна шина, как показано на рис. 8.17, *а*. Два или более процессоров и один или несколько модулей памяти используют для взаимодействия одну и ту же шину. Если процессору нужно считать слово из памяти, он сначала проверяет, свободна ли шина. Если шина свободна, процессор помещает адрес нужного слова на шину, устанавливает несколько сигналов управления и ждет, когда память поместит на шину нужное слово.

Если шина занята, процессор просто ждет, когда она освободится. С этой разработкой связана одна проблема. При наличии двух или трех процессоров доступ к шине вполне управляем; при наличии 32 и 64 процессоров возникают трудности. Производительность системы будет полностью ограничиваться пропускной способностью шины, а большинство процессоров будут простаивать большую часть времени.

Чтобы разрешить эту проблему, нужно добавить кэш-память к каждому процессору, как показано на рис. 8.17, *б*. Кэш-память может находиться внутри микросхемы процессора, рядом с микросхемой процессора, на плате процессора. Допустимы комбинации этих вариантов. Поскольку теперь считывать слова можно из кэш-памяти, движения в шине будет меньше, и система сможет поддерживать большее количество процессоров.

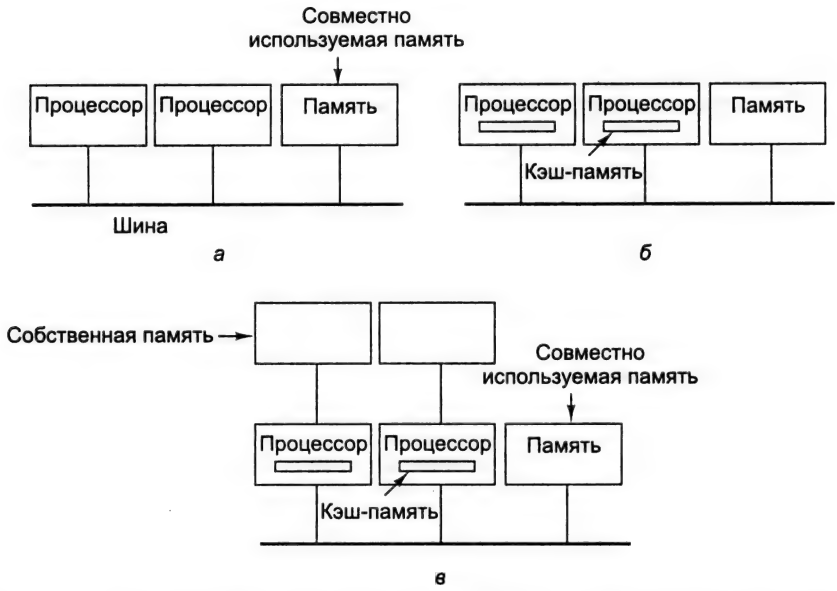


Рис. 8.17. Три мультипроцессора на одной шине: без кэш-памяти (*а*); с кэш-памятью (*б*); с кэш-памятью и отдельными блоками памяти (*в*)

Еще одна возможность — разработка, в которой каждый процессор имеет не только кэш-память, но и свою локальную память, к которой он получает доступ через назначенную локальную шину (рис. 8.17, *в*). Чтобы оптимально использовать такую конфигурацию, компилятор должен поместить в локальные модули памяти весь текст программы, цепочки, константы, другие данные, предназначенные только для чтения, стеки и локальные переменные. Общая разделенная память используется только для общих переменных. В большинстве случаев такое разумное размещение сильно сокращает количество данных, передаваемых по шине, и не требует активного вмешательства со стороны компилятора.

Отслеживание изменений данных в кэш-памяти

Предположим, что память согласована по последовательности. Что произойдет, если процессор 1 содержит в своей кэш-памяти строку, а процессор 2 пытается считать слово из той же строки кэш-памяти? При отсутствии специальных правил процессор 2 получит копию этой строки в свою кэш-память. В принципе помещение одной и той же строки в кэш-память дважды вполне приемлемо. А теперь предположим, что процессор 1 изменяет строку, и сразу после этого процессор 2 считывает копию этой строки из своей кэш-памяти. Он получит **устаревшие данные**, нарушая контракт между программным обеспечением и памятью. Ни к чему хорошему это не приведет.

Эта проблема, которую называют **непротиворечивостью кэшей**, очень важна. Если ее не разрешить, нельзя будет использовать кэш-память, и число мультипроцессоров, подсоединенных к одной шине, придется сократить до двух-трех. Специалистами было предложено множество различных решений (например, [47, 109]). Хотя все эти алгоритмы, называемые **протоколами когерентности кэширования,**

различаются в некоторых деталях, все они не допускают одновременного появления разных вариантов одной и той же строки в разных блоках кэш-памяти.

Во всех решениях контроллер кэш-памяти разрабатывается так, чтобы кэш-память могла перехватывать запросы на шине, контролируя все запросы шины от других процессоров и других блоков кэш-памяти и предпринимая те или иные действия в определенных случаях. Эти устройства называются **кэш-памятью с отслеживанием (snooping caches** или **snoopy caches)**, поскольку они отслеживают шину. Набор правил, которые выполняются кэш-памятью, процессорами и основной памятью, чтобы предотвратить появление различных вариантов данных в нескольких блоках кэш-памяти, формируют протокол когерентности кэширования. Единица передачи и хранения кэш-памяти называется строкой кэш-памяти. Обычно строка кэш-памяти равна 32 или 64 байтам.

Самый простой протокол когерентности кэширования называется **сквозным кэшированием.** Чтобы лучше понять его, рассмотрим 4 случая, приведенные в табл. 8.6. Если процессор пытается считать слово, которого нет в кэш-памяти, контроллер кэш-памяти загружает в кэш-память строку, содержащую это слово. Строку предоставляет основная память, которая в этом протоколе всегда обновлена. В дальнейшем информация может считываться из кэш-памяти.

Таблица 8.6. Сквозное кэширование. Пустые графы означают, что никакого действия не происходит

Действие	Локальный запрос	Удаленный запрос
Промах при чтении	Вызов данных из памяти	
Попадание при чтении	Использование данных из локальной кэш-памяти	
Промах при записи	Обновление данных в памяти	
Попадание при записи	Обновление кэш-памяти и основной памяти	Объявление элемента кэш-памяти недействительным

В случае промаха кэш-памяти при записи слово, которое было изменено, записывается в основную память. Строка, содержащая нужное слово, не загружается в кэш-память. В случае результативного обращения к кэш-памяти при записи кэш обновляется, а слово плюс ко всему записывается в основную память. Суть протокола состоит в том, что в результате всех операций записи записываемое слово обязательно проходит через основную память, чтобы информация в основной памяти всегда обновлялась.

Рассмотрим все эти действия снова, но теперь с точки зрения кэш-памяти с отслеживанием (крайняя правая колонка в табл. 8.6). Назовем кэш-память, которая выполняет действия, кэш-1, а кэш с отслеживанием — кэш-2. Если при считывании произошел промах кэша-1, он запрашивает шину, чтобы получить нужную строку из основной памяти. Кэш-2 видит это, но ничего не делает. Если нужная строка уже содержится в кэш-1, запроса шины не происходит, поэтому кэш-2 не знает о результативных считываниях из кэша-1.

Процесс записи более интересен. Если процессор 1 записывает слово, кэш-1 запрашивает шину как в случае промаха кэша, так и в случае попадания. Всегда при записи кэш-2 проверяет наличие у себя записываемого слова. Если данное слово

отсутствует, кэш-2 рассматривает это как промах отдаленной памяти и ничего не делает. (Отметим, что в табл. 8.6 промах отдаленной памяти означает, что слово не присутствует в кэш-памяти отслеживателя; не имеет значения, было ли это слово в кэш-памяти инициатора или нет. Таким образом, один и тот же запрос может быть результативным логически и промахом для отслеживателя и наоборот.)

А теперь предположим, что кэш-1 записывает слово, которое присутствует в кэш-2. Если кэш-2 не произведет никаких действий, он будет содержать устаревшие данные, поэтому элемент кэш-памяти, содержащий измененное слово, помечается как недействительный. Соответствующая единица просто удаляется из кэш-памяти. Все кэши отслеживают все запросы шины, и всякий раз, когда записывается слово, нужно обновить его в кэш-памяти инициатора запроса, обновить его в основной памяти и удалять его из всех других кэшей. Таким образом, неправильные варианты слова исключаются.

Процессор кэш-памяти-2 вправе прочитать то же самое слово на следующем цикле. В этом случае кэш-2 считает слово из основной памяти, которая уже обновилась. В этот момент кэш-1, кэш-2 и основная память содержат идентичные копии этого слова. Если какой-нибудь процессор произведет запись, то другие кэши будут очищены, а основная память опять обновится.

Возможны различные вариации этого основного протокола. Например, при успешной записи отслеживающий кэш обычно объявляет недействительным элемент, содержащий данное слово. С другой стороны, вместо того чтобы объявлять слово недействительным, можно принять новое значение и обновить кэш-память. По существу, обновить кэш-память — это то же самое, что признать слово недействительным, а затем считать нужное слово из основной памяти. Во всех кэш-протоколах нужно сделать выбор между **стратегией обновления** и **стратегией с признанием данных недействительными**. Эти протоколы работают по-разному. Сообщения об обновлении несут полезную нагрузку, и следовательно, они больше по размеру, чем сообщения о недействительности, но зато они могут предотвратить дальнейшие промахи кэш-памяти.

Другой вариант — загрузка отслеживающей кэш-памяти при промахах. Такая загрузка никак не влияет на правильность выполнения алгоритма. Она влияет только на производительность. Возникает вопрос: какова вероятность, что только что записанное слово вскоре будет записано снова? Если вероятность высока, то можно говорить в пользу загрузки кэш-памяти при промахах записи (**политика заполнения по записи**). Если вероятность мала, лучше не обновлять кэш-память в случае промаха при записи. Если данное слово скоро будет считываться, оно все равно будет загружено после промаха при считывании, и нет смысла загружать его в случае промаха при записи.

Как и большинство простых решений, это решение не очень эффективно. Каждая операция записи должна передаваться в основную память по шине, а при большом количестве процессоров это затруднительно. Поэтому были разработаны другие протоколы. Все они характеризуются одним общим свойством: не все записи проходят непосредственно через основную память. Вместо этого при изменении строки кэш-памяти внутри кэш-памяти устанавливается бит, который указывает, что строка в кэш-памяти правильная, а в основной памяти — нет. В конечном итоге эту строку нужно будет записать в основную память, но перед этим в память

можно произвести много записей. Такой тип протокола называется **протоколом с обратной записью.**

Протокол MESI

Один из популярных протоколов с обратной записью называется **MESI** (по первым буквам названий четырех состояний, M, E, S и I) [109]. В его основе лежит **протокол однократной записи** [47]. Протокол MESI используется в Pentium II и других процессорах для отслеживания шины. Каждый элемент кэш-памяти может находиться в одном из следующих четырех состояний:

1. Invalid — элемент кэш-памяти содержит недействительные данные.
2. Shared — несколько кэшей могут содержать данную строку; основная память обновлена.
3. Exclusive — никакой другой кэш не содержит эту строку; основная память обновлена.
4. Modified — элемент действителен; основная память недействительна; копий элемента не существует.

При загрузке процессора все элементы кэш-памяти помечаются как недействительные. При первом считывании из основной памяти нужная строка вызывается в кэш-память данного процессора и помечается как E (Exclusive), поскольку это единственная копия в кэш-памяти (рис. 8.18, *а*). При последующих считываниях процессор использует эту строку и не использует шину. Другой процессор может вызвать ту же строку и поместить ее в кэш-память, но при отслеживании исходный держатель строки (процессор 1) узнает, что он уже не единственный, и объявляет, что у него есть копия. Обе копии помечаются состоянием S (Shared) (см. рис. 8.18, *б*). При последующих чтениях кэшированных строк в состоянии S процессор не использует шину и не меняет состояние элемента.

Посмотрим, что произойдет, если процессор 2 произведет запись в строку кэш-памяти, находящуюся в состоянии S. Тогда процессор помещает сигнал о недействительности на шину, который сообщает всем другим процессорам, что нужно отбросить свои копии. Соответствующая строка переходит в состояние M (Modified) (см. рис. 8.18, *в*). Эта строка не записывается в основную память. Отметим, что если записываемая строка находится в состоянии E, никакого сигнала о недействительности на шину передавать не следует, поскольку известно, что других копий нет.

А теперь рассмотрим, что произойдет, если процессор 3 считывает эту строку. Процессор 2, который в данный момент содержит строку, знает, что копия в основной памяти недействительна, поэтому он передает на шину сигнал, чтобы процессор 3 подождал, пока он запишет строку обратно в память. Как только строка записана в основную память, процессор 3 вызывает из памяти копию этой строки, и в обоих кэшах строка помечается как S (см. рис. 8.18, *г*). Затем процессор 2 записывает эту строку снова, что делает недействительной копию в кэш-памяти процессора 3 (см. рис. 8.18, *в*).

Наконец, процессор 1 производит запись в слово в этой строке. Процессор 2 видит это и передает на шину сигнал, который сообщает процессору 1, что нужно подождать, пока строка не будет записана в основную память. Когда это действие закончится, процессор помечает собственную копию строки как недействительную, поскольку он знает, что другой процессор собирается изменить ее. Возникает

ситуация, в которой процессор записывает что-либо в некэшированную строку. Если применяется политика write-allocate, строка будет загружаться в кэш-память и помечаться как M (рис. 8.18, *е*). Если политика write-allocate не применяется, запись будет производиться непосредственно в основную память, а строка в кэш-памяти сохранена не будет.

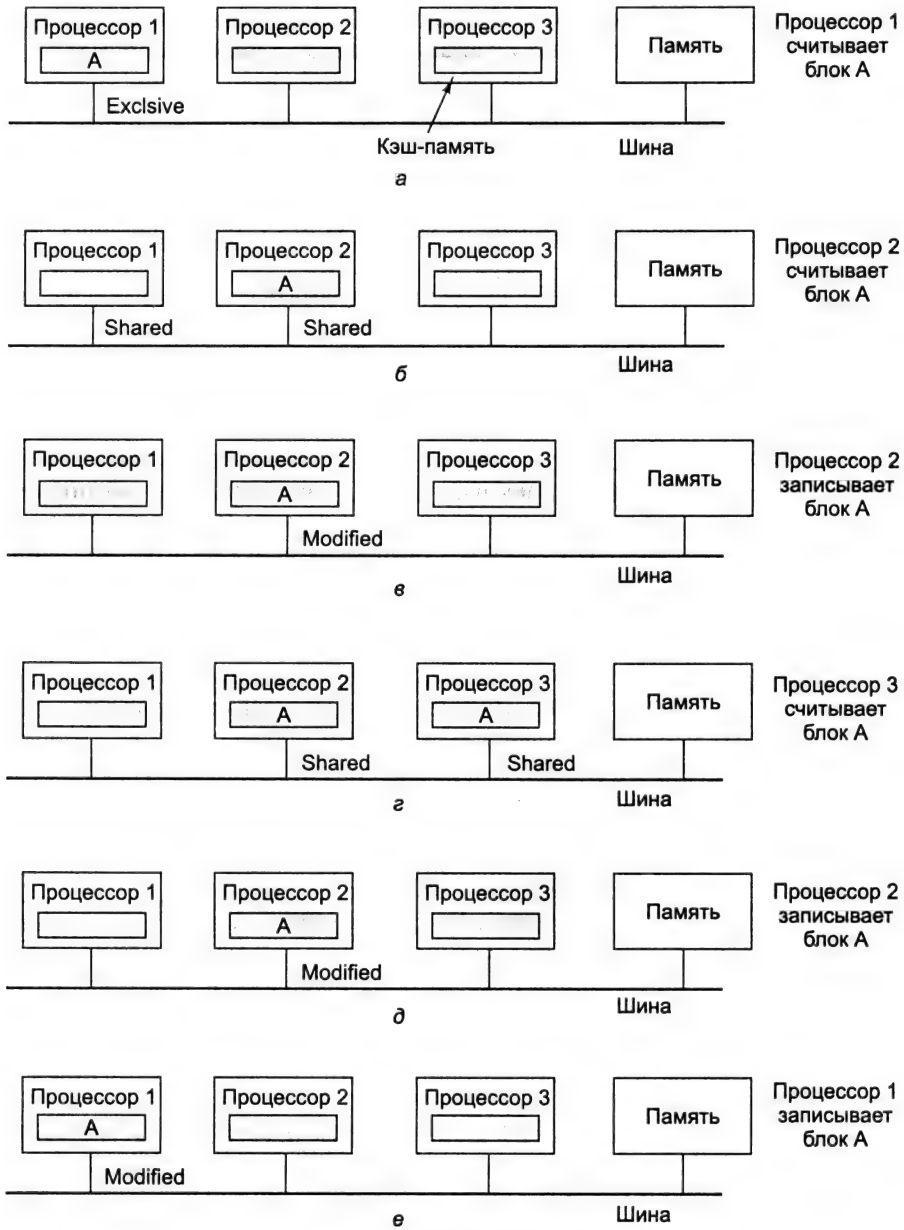


Рис. 8.18. Протокол MESI

Мультипроцессоры UMA с координатными коммутаторами

Даже при всех возможных оптимизациях использование только одной шины ограничивает размер мультипроцессора UMA до 16 или 32 процессоров. Чтобы получить больший размер, требуется другой тип коммуникационной сети. Самая простая схема соединения n процессоров с k блоками памяти — **координатный коммутатор** (рис. 8.19). Координатные коммутаторы используются на протяжении многих десятилетий для соединения группы входящих линий с рядом выходящих линий произвольным образом.

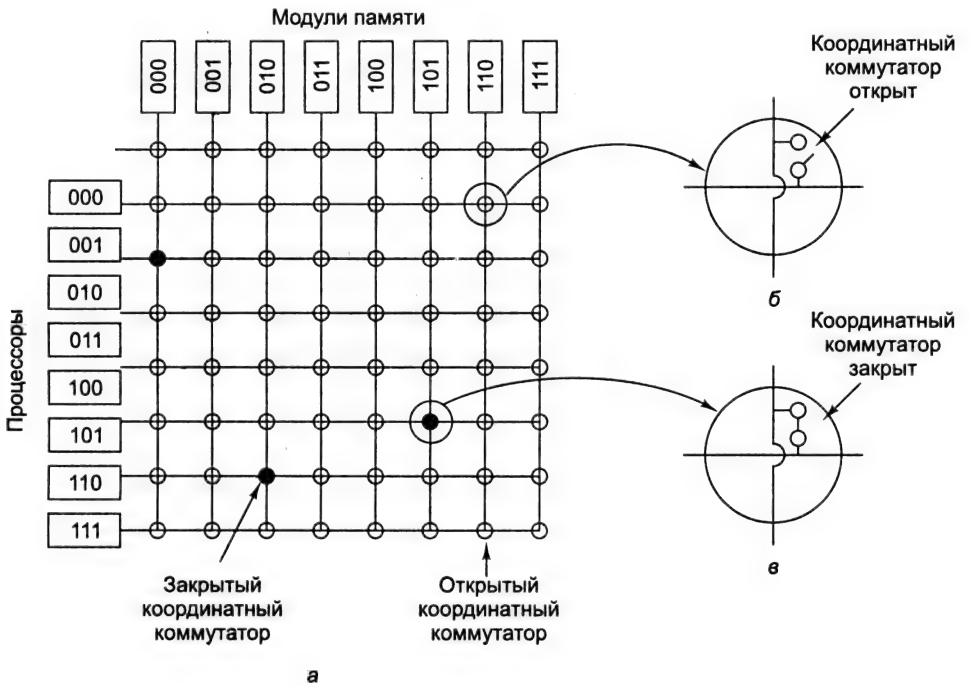


Рис. 8.19. Координатный коммутатор 8×8 (*а*); открытый узел (*б*); закрытый узел (*в*)

В каждом пересечении горизонтальной (входящей) и вертикальной (исходящей) линии находится **соединение** (crosspoint), которое можно открыть или закрыть в зависимости от того, нужно соединять горизонтальную и вертикальную линии или нет. На рис. 8.19, *а* мы видим, что три узла закрыты, благодаря чему устанавливается связь между парами (процессор, память) (001, 000), (101, 101) и (110, 010) одновременно. Возможны другие комбинации. Число комбинаций равно числу способов, которыми можно расставить 8 ладей на шахматной доске.

Координатный коммутатор представляет собой **неблокируемую сеть**. Это значит, что процессор всегда будет связан с нужным блоком памяти, даже если какая-то линия или узел уже заняты. Более того, никакого предварительного планирования не требуется. Даже если уже установлено семь произвольных связей, всегда

можно связать оставшийся процессор с оставшимся блоком памяти. Ниже мы рассмотрим схемы, которые не обладают такими свойствами.

Не лучшим свойством координатного коммутатора является то, что число узлов растет как n^2. При наличии 1000 процессоров и 1000 блоков памяти нам понадобится миллион узлов. Это неприемлемо. Тем не менее координатные коммутаторы вполне применимы для систем средних размеров.

Sun Enterprise 10000

В качестве примера мультипроцессора UMA, основанного на координатном коммутаторе, рассмотрим систему Sun Enterprise 10000 [23, 24]. Эта система состоит из одного корпуса с 64 процессорами. Координатный коммутатор **Gigaplahe-XB** запакован в плату, содержащую 8 гнезд на каждой стороне. Каждое гнездо вмещает огромную плату процессора (40×50 см), содержащую 4 процессора UltraSPARC на 333 МГц и ОЗУ на 4 Гбайт. Благодаря жестким требованиям к синхронизации и малому времени ожидания доступ к памяти вне платы занимает столько же времени, сколько доступ к памяти на плате.

Иметь только одну шину для взаимодействия всех процессоров и всех блоков памяти неудобно, поэтому в системе Enterprise 10000 применяется другая стратегия. Здесь есть координатный коммутатор 16×16 для перемещения данных между основной памятью и блоками кэш-памяти. Длина строки кэш-памяти составляет 64 байта, а ширина канала связи составляет 16 байтов, поэтому для перемещения строки кэш-памяти требуется 4 цикла. Координатный коммутатор работает от точки к точке, поэтому его нельзя использовать для сохранения совместимости по кэш-памяти.

По этой причине помимо координатного коммутатора имеются 4 адресные шины, которые используются для отслеживания строк в кэш-памяти (рис. 8.20). Каждая шина используется для 1/4 физического адресного пространства. Для выбора шины используется два адресных бита. В случае промаха кэш-памяти при считывании процессор должен считывать нужную ему информацию из основной памяти, и тогда он обращается к соответствующей адресной шине, чтобы узнать, нет ли нужной строки в других блоках кэш-памяти. Все 16 плат отслеживают все 4 адресные шины одновременно, поэтому если ответа нет, это значит, что требуемая строка отсутствует в кэш-памяти и ее нужно вызывать из основной памяти.

Вызов из памяти происходит от точки к точке по координатному коммутатору по 16 байтов. Цикл шины составляет 12 нс (83,3 МГц), и каждая адресная шина может отслеживаться в каждом цикле любой другой шины, то есть всего возможно 167 млн отслеживаний/с. Каждое отслеживание может потребовать передачи строки кэш-памяти в 64 байта, поэтому узел должен быть способен передавать 9,93 Гбайт/с (напомним, что 1 Гбайт=$1{,}0737 \times 10^9$ байт/с, а не 10^9 байт/с). Строку кэш-памяти в 64 байта можно передать через узел за 4 цикла шины (48 нс) при пропускной способности 1,24 Гбайт/с за одну передачу. Поскольку узел может обрабатывать 16 передач одновременно, его максимальная пропускная способность составляет 19,87 Гбайт/с, а этого достаточно для поддержания скорости отслеживания, даже если принять во внимание конфликтную ситуацию, которая сокращает практическую пропускную способность примерно до 60% от теоретической.

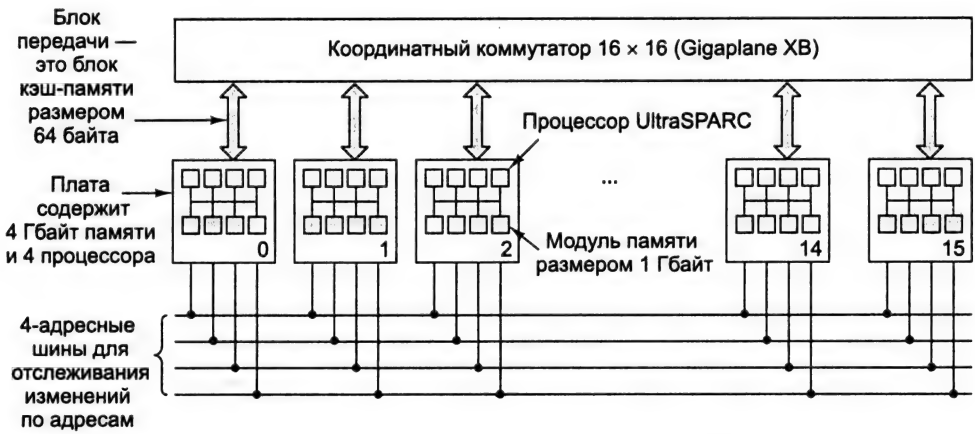


Рис. 8.20. Мультипроцессор Sun Enterprise 10000

Enterprise 10000 использует 4 отслеживающие шины параллельно, плюс очень широкий координатный коммутатор для передачи данных. Ясно, что такая система преодолевает предел в 64 процессора. Но чтобы существенно увеличить количество процессоров, требуется совсем другой подход.

Мультипроцессоры UMA с многоступенчатыми сетями

В основе «совсем другого подхода» лежит небольшой коммутатор 2×2 (рис. 8.21, *а*). Этот коммутатор содержит два входа и два выхода. Сообщения, приходящие на любую из входных линий, могут переключаться на любую выходную линию. В нашем примере сообщения будут содержать до четырех частей (рис. 8.21, *б*). Поле *Модуль* сообщает, какую память использовать. Поле *Адрес* определяет адрес в этом модуле памяти. В поле *Код операции* содержится операция, например `READ` или `WRITE`. Наконец, дополнительное поле *Значение* может содержать операнд, например 32-битное слово, которое нужно записать при выполнении операции `WRITE`. Коммутатор исследует поле *Модуль* и использует его для определения, через какую выходную линию нужно отправить сообщение: через X или через Y.

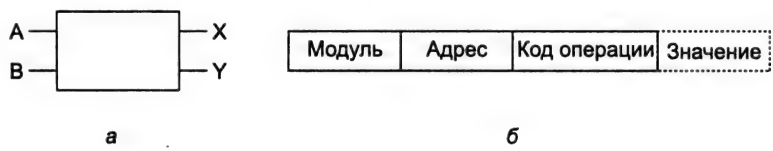


Рис. 8.21. Коммутатор 2×2 (*а*); формат сообщения (*б*)

Наши коммутаторы 2×2 можно компоновать различными способами и получать **многоступенчатые сети** [1, 15, 78]. Один из возможных вариантов — **сеть omega** (рис. 8.22). Здесь мы соединили 8 процессоров с 8 модулями памяти, используя 12 коммутаторов. Для n процессоров и n модулей памяти нам понадобится $\log_2 n$ ступеней, n/2 коммутаторов на каждую ступень, то есть всего $(n/2)\log_2 n$

коммутаторов, что намного лучше, чем n^2 узлов (точек пересечения), особенно для больших n.

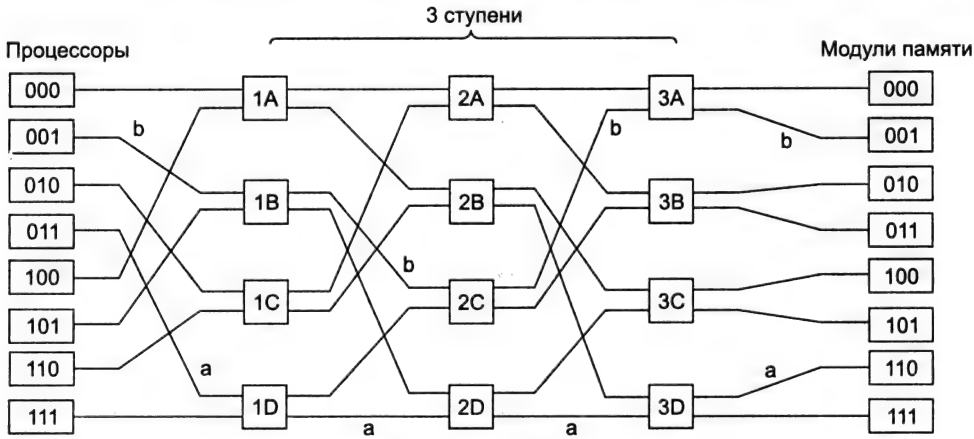


Рис. 8.22. Сеть omega

Рисунок разводки сети omega часто называют **полным тасованием**, поскольку смешение сигналов на каждой ступени напоминает колоду карт, которую разделили пополам, а затем снова соединили, чередуя карты. Чтобы понять, как работает сеть omega, предположим, что процессору 011 нужно считать слово из модуля памяти 110. Процессор посылает сообщение READ, чтобы переключить коммутатор 1D, который содержит 110 в поле *Модуль*. Коммутатор берет первый (то есть крайний левый) бит от 110 и по нему узнает направление. 0 указывает на верхний выход, а 1 — на нижний. Поскольку в данном случае этот бит равен 1, сообщение отправляется через нижний выход в 2D.

Все коммутаторы второй ступени, включая 2D, для определения направления используют второй бит. В данном случае он равен 1, поэтому сообщение отправляется через нижний выход в 3D. Затем проверяется третий бит. Он равен 0. Следовательно, сообщение переходит в верхний выход и прибывает в память 110, чего мы и добивались. Путь, пройденный данным сообщением, обозначен на рис. 8.22 буквой a.

Как только сообщение пройдет через сеть, крайние левые биты номера модуля больше не требуются. Их можно использовать, записав туда номер входной линии, чтобы было известно, по какому пути посылать ответ. Для пути a входные линии — это 0 (верхний вход в 1D), 1 (нижний вход в 2D) и 1 (нижний вход в 3D) соответственно. При отправке ответа тоже используется 011, только теперь число читается справа налево.

В то время как все это происходит, процессору 001 нужно записать слово в модуль памяти 001. Здесь происходит аналогичный процесс. Сообщение отправляется через верхний, верхний и нижний выходы соответственно. На рис. 8.22 этот путь отмечен буквой b. Когда сообщение пребывает в пункт назначения, в поле *Модуль* содержится 001. Это число показывает путь, который прошло сообщение. Поскольку эти два запроса используют совершенно разные коммутаторы, линии и модули памяти, они могут протекать параллельно.

А теперь рассмотрим, что произойдет, если процессору 000 одновременно с этим понадобился доступ к модулю памяти 000. Его запрос вступит в конфликт с запросом процессора 001 на коммутаторе 3A. Одному из них придется подождать. В отличие от координатного коммутатора, сеть omega — это **блокируемая сеть**. Не всякий набор запросов может передаваться одновременно. Конфликты могут возникать при использовании одного и того же провода или одного и того же коммутатора, а также между запросами, направленными к памяти, и ответами, исходящими из памяти.

Желательно равномерно распределить обращения к памяти по модулям. Один из возможных способов — использовать младшие биты в качестве номера модуля памяти. Рассмотрим адресное пространство с побайтовой адресацией для компьютера, который в основном получает доступ к 32-битным словам. Два младших бита обычно будут 00, но следующие три бита будут равномерно распределены. Если использовать эти три бита в качестве номера модуля памяти, последовательно адресуемые слова будут находиться в последовательных модулях. Система памяти, в которой последовательные слова находятся в разных модулях памяти, называется **расслоенной**. Расслоенная система памяти доводит параллелизм до максимума, поскольку большая часть обращений к памяти — это обращения к последовательным адресам. Можно разработать неблокируемые сети, в которых существует несколько путей от каждого процессора к каждому модулю памяти.

Мультипроцессоры NUMA

Размер мультипроцессоров UMA с одной шиной обычно ограничивается до нескольких десятков процессоров, а для координатных мультипроцессоров или мультипроцессоров с коммутаторами требуется дорогое аппаратное обеспечение, и они ненамного больше по размеру. Чтобы получить более 100 процессоров, нужно что-то предпринять. Отметим, что все модули памяти имеют одинаковое время доступа. Это наблюдение приводит к разработке мультипроцессоров **NUMA (NonUniform Memory Access — с неоднородным доступом к памяти)**. Как и мультипроцессоры UMA, они обеспечивают единое адресное пространство для всех процессоров, но, в отличие от машин UMA, доступ к локальным модулям памяти происходит быстрее, чем к удаленным. Следовательно, все программы UMA будут работать без изменений на машинах NUMA, но производительность будет хуже, чем на машине UMA с той же тактовой частотой.

Машины NUMA имеют три ключевые характеристики, которыми все они обладают и которые в совокупности отличают их от других мультипроцессоров:

1. Существует одно адресное пространство, видимое для всех процессоров.
2. Доступ к удаленной памяти производится с использованием команд LOAD и STORE.
3. Доступ к удаленной памяти происходит медленнее, чем доступ к локальной памяти.

Если время доступа к удаленной памяти не скрыто (поскольку кэш-память отсутствует), то такая система называется **NC-NUMA (No Caching NUMA — NUMA без кэширования)**. Если присутствуют согласованные кэши, то система называется **CC-NUMA (Coherent Cache NUMA — NUMA с согласованной кэш-памятью)**.

Программисты часто называют ее **аппаратной DSM (Distributed Shared Memory — распределенная совместно используемая память)**, поскольку она по сути сходна с программной DSM, но реализуется в аппаратном обеспечении с использованием страниц маленького размера.

Одной из первых машин NC-NUMA была Carnegie-Mellon Cm*. Она проиллюстрирована в упрощенной форме рис. 8.23 [143]. Машина состояла из набора процессоров LSI-11, каждый с собственной памятью, обращение к которой производится по локальной шине. (LSI-11 — это один из видов процессора DEC PDP-11 на одной микросхеме; этот мини-компьютер был очень популярен в 70-е годы.) Кроме того, системы LSI-11 были связаны друг с другом системной шиной. Когда запрос памяти приходил в блок управления памятью, производилась проверка и определялось, находится ли нужное слово в локальной памяти. Если да, то запрос отправлялся по локальной шине. Если нет, то запрос направлялся по системной шине к системе, которая содержала данное слово. Естественно, вторая операция занимала гораздо больше времени, чем первая. Выполнение программы из удаленной памяти занимало в 10 раз больше времени, чем выполнение той же программы из локальной памяти.

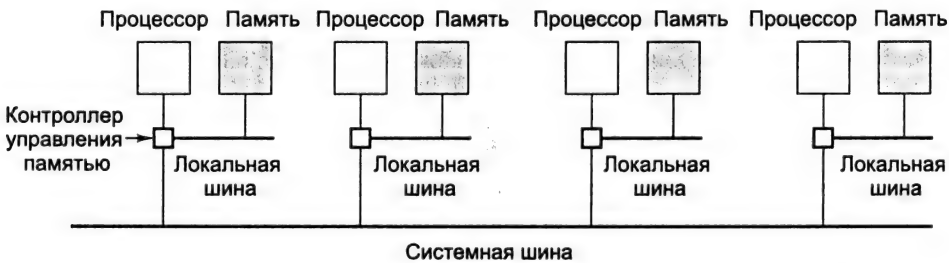


Рис. 8.23. Машина NUMA с двумя уровнями шин. Cm* — первый мультипроцессор, в котором использовалась данная разработка

Согласованность памяти гарантирована в машине NC-NUMA, поскольку там отсутствует кэш-память. Каждое слово памяти находится только в одном месте, поэтому нет никакой опасности появления копии с устаревшими данными: здесь вообще нет копий. Имеет большое значение, в какой именно памяти находится та или иная страница, поскольку от этого зависит производительность. Машины NC-NUMA используют сложное программное обеспечение для перемещения страниц, чтобы максимально увеличить производительность.

Обычно существует «сторожевой» процесс (демон), так называемый страничный сканер, который запускается каждые несколько секунд. Он должен следить за статистикой использования страниц и перемещать их таким образом, чтобы улучшить производительность. Если страница окажется в неправильном месте, страничный сканер преобразует ее таким образом, чтобы следующее обращение к ней вызвало ошибку из-за отсутствия страницы. Когда происходит такая ошибка, принимается решение о том, куда поместить эту страницу, возможно, в другую память, из которой она была взята раньше. Для предотвращения пробуксовки существует правило, которое гласит, что если страница была помещена в то или иное место, она должна оставаться в этом месте на время ΔT. Было рассмотрено

множество алгоритмов, но ни один из них не работает лучше других при любых обстоятельствах [80].

Мультипроцессоры CC-NUMA

Мультипроцессоры, подобные тому, который изображен на рис. 8.23, плохо расширяются, поскольку в них нет кэш-памяти. Каждый раз переходить к удаленной памяти, чтобы получить доступ к слову, которого нет в локальной памяти, очень невыгодно: это сильно снижает производительность. Однако с добавлением кэш-памяти нужно будет добавить и способ совместимости кэшей. Один из способов — отслеживать системную шину. Технически это сделать несложно, но мы уже видели (когда рассматривали Enterprise 10000), что даже с четырьмя отслеживающими шинами и высокоскоростным координатным коммутатором шириной 16 байтов для передачи данных 64 процессора — это верхний предел. Для создания мультипроцессоров действительно большого размера нужен совершенно другой подход.

Самый популярный подход для построения больших мультипроцессоров **CC-NUMA (Cache Coherent NUMA — NUMA с согласованной кэш-памятью) — мультипроцессор на основе каталога.** Основная идея состоит в сохранении базы данных, которая сообщает, где именно находится каждая строка кэш-памяти и каково ее состояние. При обращении к строке кэш-памяти из базы данных выявляется информация о том, где находится эта строка и изменялась она или нет. Поскольку обращение к базе данных происходит на каждой команде, которая обращается к памяти, база данных должна находиться в высокоскоростном специализированном аппаратном обеспечении, которое способно выдавать ответ на запрос за долю цикла шины.

Чтобы лучше понять, что собой представляет мультипроцессор на основе каталога, рассмотрим в качестве примера систему из 256 узлов, в которой каждый узел состоит из одного процессора и 16 Мбайт ОЗУ, связанного с процессором через локальную шину. Общий объем памяти составляет 2^{32} байтов. Она разделена на 2^{26} строк кэш-памяти по 64 байта каждая. Память статически распределена по узлам: 0–16 М в узле 0, 16 М–32 М — в узле 1 и т. д. Узлы связаны через сеть (рис. 8.24, *а*). Сеть может быть в виде решетки, гиперкуба или другой топологии. Каждый узел содержит элементы каталога для 2^{18} 64-байтных строк кэш-памяти, составляя свою 2^{24}-байтную память. На данный момент мы предполагаем, что строка может содержаться максимум в одной кэш-памяти.

Чтобы понять, как работает каталог, проследим путь команды `LOAD` из процессора 20, который обращается к кэшированной строке. Сначала процессор, выдавший команду, передает ее в блок управления памятью, который переводит ее в физический адрес, например 0x24000108. Блок управления памятью разделяет этот адрес на три части, как показано на рис. 8.24, *б*. В десятичной системе счисления эти три части — узел 36, строка 4 и смещение 8. Блок управления памятью видит, что слово памяти, к которому производится обращение, находится в узле 36, а не в узле 20, поэтому он посылает запрос через сеть в узел 36, где находится нужная строка, узнает, есть ли строка 4 в кэш-памяти, и если да, то где именно.

Когда запрос прибывает в узел 36, он направляется в аппаратное обеспечение каталога. Аппаратное обеспечение индексирует таблицу их 2^{18} элементов (один

элемент на каждую строку кэш-памяти) и извлекает элемент 4. Из рис. 8.24, *в* видно, что эта строка отсутствует в кэш-памяти, поэтому аппаратное обеспечение вызывает строку 4 из локального ОЗУ, отправляет ее в узел 20 и обновляет элемент каталога 4, чтобы показать, что эта строка находится в кэш-памяти в узле 20.

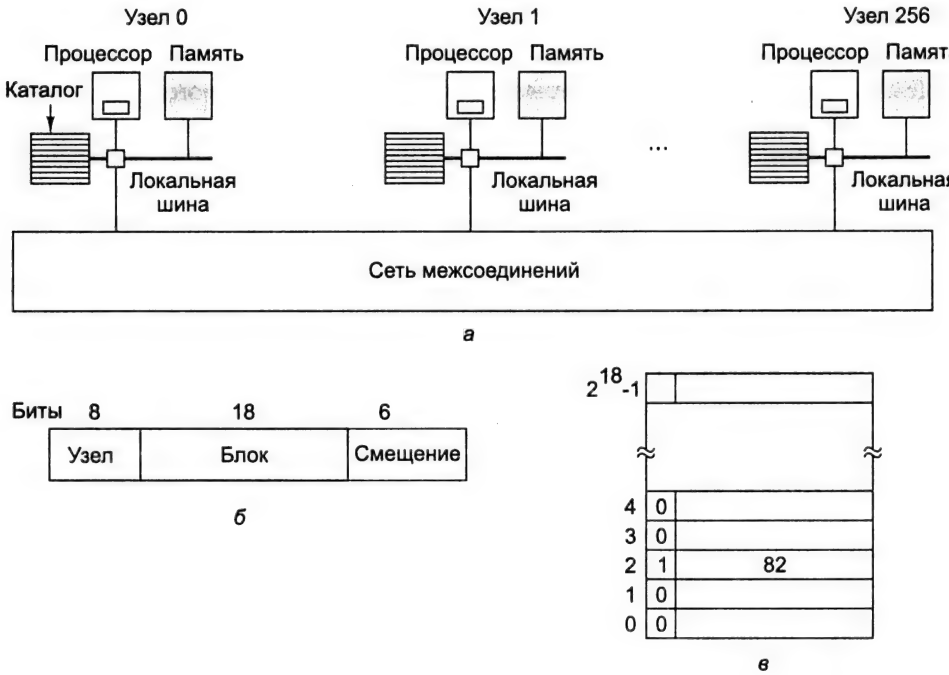


Рис. 8.24. Мультипроцессор на основе каталога, содержащий 256 узлов (*а*); разбиение 32-битного адреса памяти на поля (*б*); каталог в узле 36 (*в*)

А теперь рассмотрим второй запрос, на этот раз о строке 2 из узла 36. Из рис. 8.24, *в* видно, что эта строка находится в кэш-памяти в узле 82. В этот момент аппаратное обеспечение может обновить элемент каталога 2, чтобы сообщить, что строка находится теперь в узле 20, а затем может послать сообщение в узел 82, чтобы строка из него была передана в узел 20, и объявить недействительной его кэш-память. Отметим, что даже в так называемом мультипроцессоре с памятью совместного использования перемещение многих сообщений проходит скрыто.

Давайте вычислим, сколько памяти занимают каталоги. Каждый узел содержит 16 Мбайт ОЗУ и 2^{18} 9-битных элементов для слежения за этим ОЗУ. Таким образом, непроизводительные затраты каталога составляют примерно 9×2^{18} битов от 16 Мбайт или около 1,76%, что вполне допустимо. Даже если длина строки кэш-памяти составляет 32 байта, непроизводительные затраты составят всего 4%. Если длина строки кэш-памяти равна 128 байтов, непроизводительные затраты будут ниже 1%.

Очевидным недостатком этой разработки является то, что строка может быть кэширована только в одном узле. Чтобы строки можно было кэшировать в нескольких узлах, потребуется какой-то способ их нахождения (например, чтобы объявлять недействительными или обновлять их при записи). Возможны различные варианты.

Одна из возможностей — предоставить каждому элементу каталога k полей для определения других узлов, что позволит сохранять каждую строку в нескольких блоках кэш-памяти (допустимо до k различных узлов). Вторая возможность — заменить номер узла битовым отображением, один бит на узел. Здесь нет ограничений на количество копий, но существенно растут непроизводительные затраты. Каталог, содержащий 256 битов для каждой 64-байтной (512-битной) строки кэш-памяти, подразумевает непроизводительные затраты выше 50%. Третья возможность — хранить в каждом элементе каталога 8-битное поле и использовать это поле как заголовок связанного списка, который связывает все копии строки кэш-памяти вместе. При такой стратегии требуется дополнительное пространство в каждом узле для указателей связанного списка. Кроме того, требуется просматривать связанный список, чтобы в случае необходимости найти все копии. Каждая из трех стратегий имеет свои преимущества и недостатки. На практике используются все три стратегии.

Еще одна проблема данной разработки — как следить за тем, обновлена ли исходная память или нет. Если нужно считать строку кэш-памяти, которая не изменялась, запрос может быть удовлетворен из основной памяти, и при этом не нужно направлять запрос в кэш-память. Если нужно считать строку кэш-памяти, которая была изменена, то этот запрос должен быть направлен в тот узел, в котором находится нужная строка кэш-памяти, поскольку только здесь имеется действительная копия. Если разрешается иметь только одну копию строки кэш-памяти, как на рис. 8.24, то нет никакого смысла в отслеживании изменений в строках кэш-памяти, поскольку любой новый запрос должен пересылаться к существующей копии, чтобы объявить ее недействительной.

Когда строка кэш-памяти меняется, нужно сообщить в исходный узел, даже если существует только одна копия строки кэш-памяти. Если существует несколько копий, изменение одной из них требует объявления всех остальных недействительными. Поэтому нужен какой-либо протокол, чтобы устранить ситуацию состояния гонок. Например, чтобы изменить общую строку кэш-памяти, один из держателей этой строки должен запросить монопольный доступ к ней перед тем, как изменить ее. В результате все другие копии будут объявлены недействительными. Другие возможные оптимизации CC-NUMA обсуждаются в книге [140].

Мультипроцессор Stanford DASH

Первый мультипроцессор CC-NUMA на основе каталога — **DASH (Directory Architecture for SHared memory — архитектура на основе каталога для памяти совместного использования)** — был создан в Стенфордском университете как исследовательский проект [81]. Данная разработка проста для понимания. Она повлияла на ряд промышленных изделий, например SGI Origin 2000. Мы рассмотрим 64-процессорный прототип данной разработки, который был реально сконструирован. Он подходит и для машин большего размера.

Схема машины DASH в немного упрощенном виде представлена на рис. 8.25, *а*. Она состоит из 16 кластеров, каждый из которых содержит шину, 4 процессора MIPS R3000, 16 Мбайт глобальной памяти, а также некоторые устройства ввода-вывода (диски и т. д.), которые на схеме не показаны. Каждый процессор отслеживает только свою локальную шину. Локальная совместимость поддерживается

с помощью отслеживания; для глобальной согласованности нужен другой механизм, поскольку глобального отслеживания не существует.

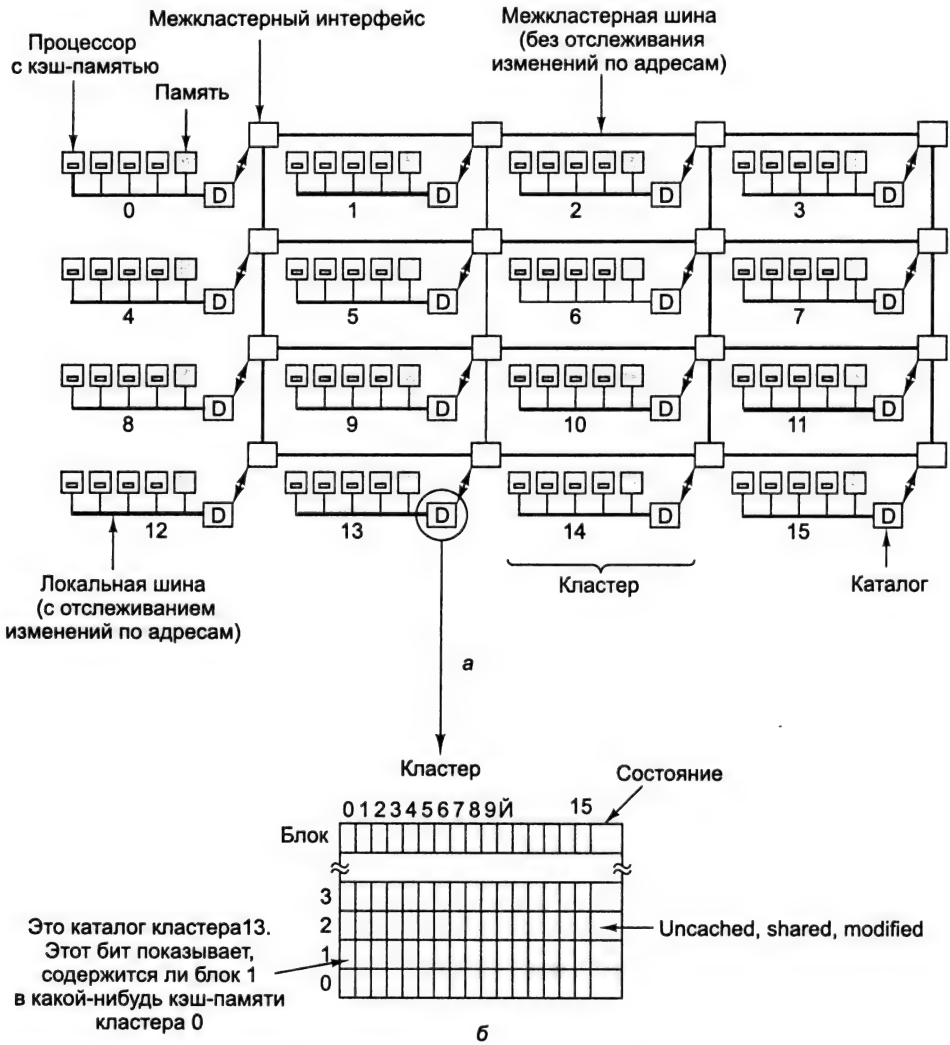


Рис. 8.25. Архитектура DASH (*а*); каталог DASH (*б*)

Полный объем адресного пространства в данной системе равен 256 Мбайт. Адресное пространство разделено на 16 областей по 16 Мбайт каждая. Глобальная память кластера 0 включает адреса с 0 по 16 М. Глобальная память кластера 1 включает адреса с 16 М по 32 М и т. д. Размер строки кэш-памяти составляет 16 байтов. Передача данных также осуществляется по строкам в 16 байтов. Каждый кластер содержит 1 М строк.

Каждый кластер содержит каталог, который следит за тем, какие кластеры в настоящий момент имеют копии своих строк. Поскольку каждый кластер содержит

1 М строк, в каталоге содержится 1 М элементов, по одному элементу на каждую строку. Каждый элемент содержит битовое отображение по одному биту на кластер. Этот бит показывает, имеется ли в данный момент строка данного кластера в кэш-памяти. Кроме того, элемент содержит 2-битное поле, которое сообщает о состоянии строки.

1 М элементов по 18 битов каждый означает, что общий размер каждого каталога превышает 2 Мбайт. При наличии 16 кластеров вся память каталога будет немного превышать 36 Мбайт, что составляет около 14% от 256 Мбайт. Если число процессоров на кластер возрастает, объем памяти каталога не меняется. Большое число процессоров на каждый кластер позволяет погашать стоимость памяти каталога, а также контроллера шины при наличии большого числа процессоров, сокращая стоимость на каждый процессор. Именно поэтому каждый кластер имеет несколько процессоров.

Каждый кластер в DASH связан с интерфейсом, который дает возможность кластеру обмениваться информацией с другими кластерами. Интерфейсы связаны через межкластерные каналы в прямоугольную решетку, как показано на рис. 8.25, *а*. Чем больше добавляется кластеров в систему, тем больше нужно добавлять межкластерных каналов, поэтому пропускная способность системы возрастает. В системе используется маршрутизация «червоточина», поэтому первая часть пакета может быть направлена дальше еще до того, как получен весь пакет, что сокращает задержку в каждом транзитном участке. Существует два набора межкластерных каналов: один — для запрашивающих пакетов, а другой — для ответных пакетов (на рисунке это не показано). Межкластерные каналы нельзя отслеживать.

Каждая строка кэш-памяти может находиться в одном из трех следующих состояний:

1. UNCACHED (некэшированная) — строка находится только в памяти.
2. SHARED (совместно используемая) — память содержит новейшие данные; строка может находиться в нескольких блоках кэш-памяти.
3. MODIFIED (измененная) — строка, содержащаяся в памяти, неправильная; данная строка находится только в одной кэш-памяти.

Состояние каждой строки кэш-памяти содержится в поле *Состояние* в соответствующем элементе каталога, как показано на рис. 8.25, *б*.

Протоколы DASH основаны на обладании и признании недействительности. В каждый момент у каждой строки имеется уникальный владелец. Для строк в состоянии UNCHANGED или SHARED владельцем является собственный кластер данной строки. Для строк в состоянии MODIFIED владельцем является тот кластер, в котором содержится единственная копия этой строки. Прежде чем записать что-либо в строку в состоянии SHARED, нужно найти и объявить недействительными все существующие копии.

Чтобы понять, как работает этот механизм, рассмотрим, как процессор считывает слово из памяти. Сначала он проверяет свою кэш-память. Если там слова нет, на локальную шину кластера передается запрос, чтобы узнать, содержит ли какой-нибудь другой процессор того же кластера строку, в которой присутствует нужное слово. Если да, то происходит передача строки из одной кэш-памяти в другую. Если строка находится в состоянии SHARED, то создается ее копия. Если строка нахо-

дится в состоянии MODIFIED, нужно проинформировать исходный каталог, что строка теперь SHARED. В любом случае слово берется из какой-то кэш-памяти, но это не влияет на битовое отображение каталогов (поскольку каталог содержит 1 бит на кластер, а не 1 бит на каждый процессор).

Если нужная строка не присутствует ни в одной кэш-памяти данного кластера, то пакет с запросом отправляется в исходный кластер, содержащий данную строку. Этот кластер определяется по 4 старшим битам адреса памяти, и им вполне может оказаться кластер запрашивающей стороны. В этом случае сообщение физически не посылается. Аппаратное обеспечение в исходном кластере проверяет свои таблицы и выясняет, в каком состоянии находится строка. Если она UNCACHED или SHARED, аппаратное обеспечение, которое управляет каталогом, вызывает эту строку из глобальной памяти и посылает ее обратно в запрашивающий кластер. Затем аппаратное обеспечение обновляет свой каталог, помечая данную строку как сохраненную в кэш-памяти кластера запрашивающей стороны.

Если нужная строка находится в состоянии MODIFIED, аппаратное обеспечение находит кластер, который содержит эту строку, и посылает запрос туда. Затем кластер, который содержит данную строку, посылает ее в запрашивающий кластер и помечает ее копию как SHARED, поскольку теперь эта строка находится более чем в одной кэш-памяти. Он также посылает копию обратно в исходный кластер, чтобы обновить память и изменить состояние строки на SHARED.

Запись происходит по-другому. Перед тем как осуществить запись, процессор должен убедиться, что он является единственным обладателем данной строки кэш-памяти в системе. Если в кэш-памяти данного процессора уже есть эта строка и она находится в состоянии MODIFIED, то запись можно осуществить сразу же. Если строка в кэш-памяти есть, но она находится в состоянии SHARED, то сначала в исходный кластер посылается пакет, чтобы объявить все остальные копии недействительными.

Если нужной строки нет в кэш-памяти данного процессора, этот процессор посылает запрос на локальную шину, чтобы узнать, нет ли этой строки в соседних процессорах. Если данная строка там есть, то она передается из одной кэш-памяти в другую. Если эта строка SHARED, то все остальные копии должны быть объявлены недействительными.

Если строка находится где-либо еще, пакет посылается в исходный кластер. Здесь может быть три варианта. Если строка находится в состоянии UNCACHED, она помечается как MODIFIED и отправляется к запрашивающему процессору. Если строка находится в состоянии SHARED, все копии объявляются недействительными, и после этого над строкой совершается та же процедура, что и над UNCASHED строкой. Если строка находится в состоянии MODIFIED (изменена), то запрос направляется в тот кластер, в котором строка содержится в данный момент. Этот кластер удовлетворяет запрос, а затем объявляет недействительной свою собственную копию.

Сохранить согласованность памяти в системе DASH довольно трудно, и происходит это очень медленно. Для одного обращения к памяти порой нужно отправлять большое количество сообщений. Более того, чтобы память была согласованной, доступ нельзя завершить, пока прием всех пакетов не будет подтвержден, а это плохо влияет на производительность. Для разрешения этих проблем в систе-

ме DASH используется ряд специальных приемов (это могут быть два набора межкластерных каналов, конвейеризированные записи, а также использование свободной согласованности вместо согласованности по последовательности).

Мультипроцессор Sequent NUMA-Q

Машина DASH никогда не была коммерческим продуктом. В этом разделе мы рассмотрим одно из коммерческих изделий — машину Sequent NUMA-Q 2000. В ней используется очень интересный протокол когерентности кэширования — **SCI (Scalable Coherent Interface — масштабируемый когерентный интерфейс)**. Этот протокол стандартный (стандарт IEEE 1569), поэтому он используется и в ряде других машин CC-NUMA.

В основе машины NUMA-Q лежит стандартная плата **quard board**, которая произведена компанией Intel. Плата содержит 4 процессора Pentium Pro и до 4 Гбайт ОЗУ. Каждый процессор содержит кэш-память первого уровня и кэш-память второго уровня. Непротиворечивость кэшей сохраняется благодаря отслеживанию локальной шины платы quard board с использованием протокола MESI. Скорость передачи данных в локальной шине составляет 534 Мбайт/с. Размер строки кэш-памяти равен 64 байтам. Схема мультипроцессора NUMA-Q изображена на рис. 8.26.

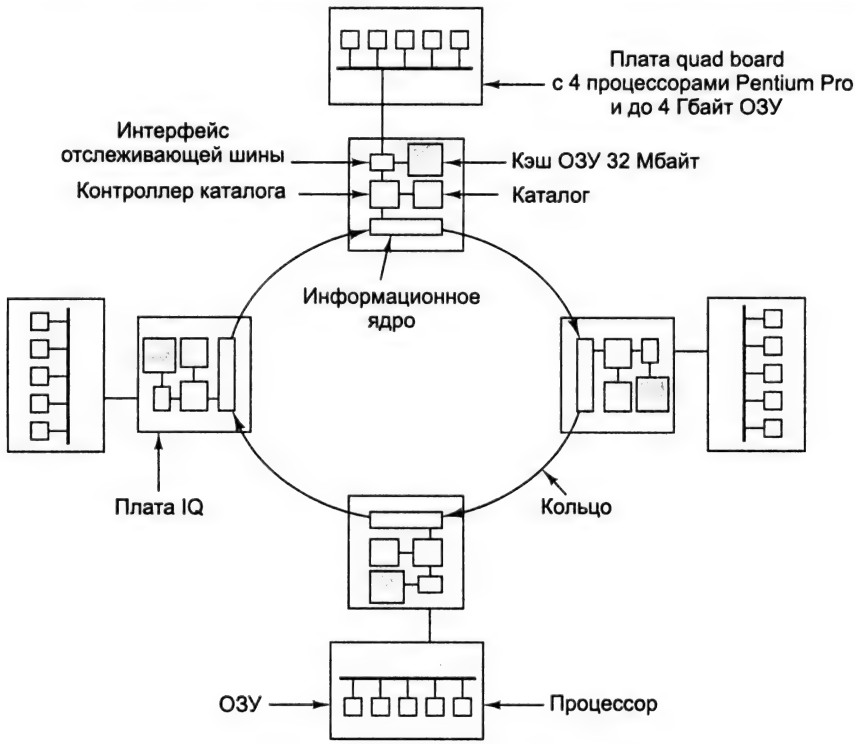


Рис. 8.26. Мультипроцессор NUMA-Q

Чтобы расширить систему, нужно вставить плату сетевого контроллера в гнездо платы quad board, предназначенное для контроллеров сети. Сетевой контроллер,

плата IQ-Link, соединяет все платы quad board в один мультипроцессор. Ее главная задача — реализовать протокол SCI. Каждая плата IQ-Link содержит 32 Мбайт кэш-памяти, каталог, который следит за тем, что находится в кэш-памяти, интерфейс с локальной шиной платы quad board и микросхему, называемую **информационным ядром**, соединяющую плату IQ-Link с другими платами IQ-Link. Эта микросхема подкачивает данные от входа к выходу, сохраняя те данные, которые направляются в данный узел, и передавая все прочие данные далее без изменений.

Все платы IQ-Link в совокупности формируют кольцо, как показано на рис. 8.26. В данной разработке присутствует два уровня протокола когерентности кэширования. Протокол SCI поддерживает непротиворечивость всех кэшей платы IQ-Link, используя это кольцо. Протокол MESI используется для сохранения непротиворечивости между четырьмя процессорами и кэш-памятью на 32 Мбайт в каждом узле.

В качестве связи между платами quad board используется интерфейс SCI. Этот интерфейс был разработан для того, чтобы заменить шину в больших мультипроцессорах и мультикомпьютерах (например, NUMA-Q). SCI поддерживает непротиворечивость кэшей, которая необходима в мультипроцессорах, а также позволяет быстро передавать блоки, что необходимо в мультикомпьютерах. SCI выдерживает нагрузку до 64 K узлов, адресное пространство каждого из которых может быть до 2^{48} байтов. Самая большая система NUMA-Q состоит из 63 плат quad board, которые содержат 252 процессора и почти 2^{38} байтов физической памяти. Как видим, возможности SCI гораздо выше.

Кольцо, которое соединяет платы IQ-Link, соответствует протоколу SCI. В действительности это вообще не кольцо, а отдельные двухточечные кабели. Ширина кабеля составляет 18 битов: 1 бит синхронизации, 1 флаговый бит и 16 битов данных. Все они передаются параллельно. Каналы синхронизируются с тактовой частотой 500 МГц, при этом скорость передачи данных составляет 1 Гбайт/с. По каналам передаются пакеты. Каждый пакет содержит заголовок из 14 байтов, 0, 16, 64 или 256 байтов данных и контрольную сумму на 2 байта. Трафик состоит из запросов и ответов.

Физическая память в машине NUMA-Q 2000 распределена по узлам, так что каждая страница памяти имеет свою собственную машину. Каждая плата quad board может вмещать до 4 Гбайт ОЗУ. Размер строки кэш-памяти равен 64 байтам, поэтому каждая плата quad board содержит 2^{26} строк кэш-памяти. Когда строка не используется, она находится только в одном месте — в собственной памяти.

Однако строки могут находиться в нескольких разных кэшах, поэтому для каждого узла должна существовать **таблица локальной памяти** из 2^{26} элементов, по которой можно находить местоположение строк. Один из возможных вариантов — иметь на каждый элемент таблицы битовое отображение, которое показывает, какие платы IQ-Link содержат эту строку. Но в SCI не используется такое битовое отображение, поскольку оно плохо расширяется. (Напомним, что SCI может выдерживать нагрузку до 64 K узлов, и иметь 2^{26} элементов по 64 K битов каждый было бы слишком накладно.)

Вместо этого все копии строки кэш-памяти собираются в дважды связанный список. Элемент в таблице локальной памяти исходного узла показывает, в каком узле содержится головная часть списка. В машине NUMA-Q 2000 достаточно 6-битного номера, поскольку здесь может быть максимум 63 узла. Для системы SCI максимального размера достаточно будет 16-битного номера. Такая схема подходит

для больших систем гораздо лучше, чем битовое отображение. Именно это свойство делает SCI более расширяемой по сравнению с системой DASH.

Кроме таблицы локальной памяти, каждая плата IQ-Link содержит каталог с одним элементом для каждой строки кэш-памяти, которую плата в данный момент содержит. Поскольку размер кэш-памяти составляет 32 Мбайт, а строка кэш-памяти включает 64 байта, каждая плата IQ-Link может содержать до 2^{19} строк кэш-памяти. Поэтому каждый каталог содержит 2^{19} элементов, по одному элементу на каждую строку кэш-памяти.

Если строка находится только в одной кэш-памяти, то тот узел, в котором находится строка, указывается в таблице локальной памяти исходного узла. Если после этого данная строка появится в кэш-памяти другого узла, то в соответствии с новым протоколом исходный каталог будет указывать на новый элемент, который, в свою очередь, указывает на старый элемент. Таким образом формируется двухэлементный список. Все новые узлы, содержащие ту же строку, прибавляются к началу списка. Следовательно, все узлы, которые содержат эту строку, связываются в сколь угодно длинный список. На рис. 8.27 проиллюстрирован этот процесс (в данном случае строка кэш-памяти содержится в узлах 4, 9 и 22).

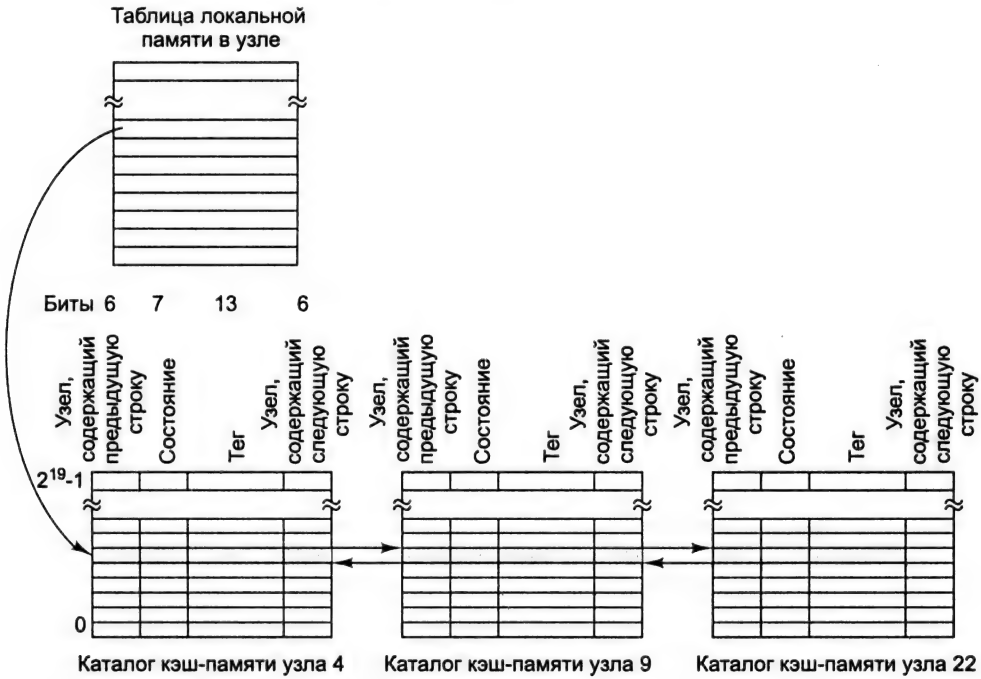


Рис. 8.27. Протокол SCI соединяет всех держателей данной строки в дважды связанный список. В данном примере строка находится одновременно в трех узлах

Каждый элемент каталога состоит из 36 битов. Шесть битов указывают на узел, который содержит предыдущую строку цепочки. Следующие шесть битов указывают на узел, содержащий следующую строку цепочки. Ноль указывает на конец цепочки, и именно поэтому максимальный размер системы составляет 63 узла, а не 64. Следующие 7 битов предназначены для записи состояния строки.

Последние 13 битов — это тег (нужен для идентификации строки). Напомним, что самая большая система NUMA-Q 2000 содержит $63 \times 2^{32} \approx 2^{38}$ байтов ОЗУ, поэтому имеется почти 2^{32} строк кэш-памяти. Все кэши прямого отображения, поэтому 2^{32} строк отображаются на 2^{19} элементов кэш-памяти, существует 2^{13} строк, которые отображаются на каждый элемент. Следовательно, 13-битный тег нужен для того, чтобы показывать, какая именно строка находится там в данный момент.

Каждая строка имеет фиксированную позицию только в одном блоке памяти (исходном). Эти строки могут находиться в одном из трех состояний: UNCACHED (некэшированном), SHARED (совместного использования) и MODIFIED (измененном). В протоколе SCI эти состояния называются HOME, FRESH и GONE соответственно, но мы во избежание путаницы будем использовать прежнюю терминологию. Состояние UNCACHED означает, что строка не содержится ни в одном из кэшей на плате IQ-Link, хотя она может находиться в локальной кэш-памяти на той же самой плате quad board. Состояние SHARED означает, что строка находится по крайней мере в одной кэш-памяти платы IQ-Link, а память содержит обновленные данные. Состояние MODIFIED означает, что строка находится в кэш-памяти на какой-то плате IQ-Link, но, возможно, эта строка была изменена, поэтому память может содержать устаревшие данные.

Блоки кэш-памяти могут находиться в одном из 29 устойчивых состояний или в одном из переходных состояний (их более 29). Для записи состояния каждой строки необходимо 7 битов (рис. 8.27). Каждое устойчивое состояние показывается двумя полями. Первое поле указывает, где находится строка — в начале списка, в конце, в середине или вообще является единственным элементом списка. Второе поле содержит информацию о том, изменялась ли строка, находится ли она только в этой кэш-памяти и т. п.

В протоколе SCI определены 3 операции со списком: добавление узла к списку, удаление узла из списка и очистка всех узлов кроме одного. Последняя операция нужна в том случае, если разделяемая строка изменена и становится единственной.

Протокол SCI имеет три варианта сложности. Протокол минимальной степени сложности разрешает иметь только одну копию каждой строки в кэш-памяти (см. рис. 8.24). В соответствии с протоколом средней степени сложности каждая строка может кэшироваться в неограниченном количестве узлов. Полный протокол включает различные особенности для увеличения производительности. В системе NUMA-Q используется протокол средней степени сложности, поэтому ниже мы рассмотрим именно его.

Разберемся, как обрабатывается команда READ. Если процессор, выполняющий эту команду, не может найти нужную строку на своей плате, то плата IQ-Link посылает пакет исходной плате IQ-Link, которая затем смотрит на состояние этой строки. Если состояние UNCACHED, оно превращается в SHARED, и нужная строка берется из основной памяти. Затем таблица локальной памяти в исходном узле обновляется. После этого таблица содержит одноэлементный список, указывающий на узел, в кэш-памяти которого в данный момент находится строка.

Предположим, что состояние строки SHARED. Эта строка также берется из памяти, и ее номер сохраняется в элементе каталога в исходном узле. Элемент каталога в запрашивающем узле устанавливается на другое значение — чтобы указывать на старый узел. Эта процедура увеличивает список на один элемент. Новая кэш-память становится первым элементом.

Представим теперь, что требуемая строка находится в состоянии MODIFIED. Исходный каталог не может вызвать эту строку из памяти, поскольку в памяти содержится недействительная копия. Вместо этого исходный каталог сообщает запрашивающей плате IQ-Link, в какой кэш-памяти находится нужная строка, и отдает приказ вызвать строку оттуда. Каталог также изменяет элемент таблицы локальной памяти, поскольку теперь он должен указывать на новое местоположение строки.

Обработка команды WRITE происходит немного по-другому. Если запрашивающий узел уже находится в списке, он должен удалять все остальные элементы, чтобы остаться в списке единственным. Если его нет в списке, он должен удалить все элементы, а затем войти в список в качестве единственного элемента. В любом случае в конце концов этот элемент остается единственным в списке, а исходный каталог указывает на этот узел. В действительности обработка команд READ и WRITE довольно сложна, поскольку протокол должен работать правильно, даже если несколько машин одновременно выполняют несовместимые операции на одной линии. Все переходные состояния были введены именно для того, чтобы протокол работал правильно даже во время одновременно выполняемых операций. В этой книге мы не можем изложить полное описание протокола. Если вам это необходимо, обратитесь к стандарту IEEE 1596.

Мультипроцессоры COMA

Машины NUMA и CC-NUMA имеют один большой недостаток: обращения к удаленной памяти происходят гораздо медленнее, чем обращения к локальной памяти. В машине CC-NUMA эта разница в производительности в какой-то степени нейтрализуется благодаря использованию кэш-памяти. Однако если количество требуемых удаленных данных сильно превышает вместимость кэш-памяти, промахи будут происходить постоянно и производительность станет очень низкой.

Мы видим, что машины UMA, например Sun Enterprise 10000, имеют очень высокую производительность, но ограничены в размерах и довольно дорого стоят. Машины NUMA могут расширяться до больших размеров, но в них требуется ручное или полуавтоматическое размещение страниц, а оно не всегда проходит удачно. Дело в том, что очень трудно предсказать, где какие страницы могут понадобиться, и кроме того, страницы трудно перемещать из-за большого размера. Машины CC-NUMA, например Sequent NUMA-Q, могут работать с очень низкой производительностью, если большому числу процессоров требуется много удаленных данных. Так или иначе, каждая из этих разработок имеет существенные недостатки.

Однако существует процессор, в котором все эти проблемы разрешаются за счет того, что основная память каждого процессора используется как кэш-память. Такая разработка называется **COMA (Cache Only Memory Access)**. В ней страницы не имеют собственных фиксированных машин, как в системах NUMA и CC-NUMA.

Вместо этого физическое адресное пространство делится на строки, которые перемещаются по системе в случае необходимости. Блоки памяти не имеют собственных машин. Память, которая привлекает строки по мере необходимости, называется **attraction memory**. Использование основной памяти в качестве большой кэш-памяти увеличивает частоту успешных обращений в кэш-память, а следовательно, и производительность.

К сожалению, ничего идеального не бывает. В системе COMA появляется две новых проблемы:

1. Как размещаются строки кэш-памяти?
2. Если строка удаляется из памяти, что произойдет, если это последняя копия?

Первая проблема связана со следующим фактом. Если блок управления памятью транслировал виртуальный адрес в физический и если строки нет в аппаратной кэш-памяти, то очень трудно определить, есть ли вообще эта строка в основной памяти. Аппаратное обеспечение здесь не поможет, поскольку каждая страница состоит из большого количества отдельных строк кэш-памяти, которые перемещаются в системе независимо друг от друга. Даже если известно, что строка отсутствует в основной памяти, как определить, где она находится? В данном случае нельзя спросить об этом собственную машину, поскольку таковой машины в системе нет.

Было предложено несколько решений этой проблемы. Можно ввести новое аппаратное обеспечение, которое будет следить за тегом каждой строки кэш-памяти. Тогда блок управления памятью может сравнивать тег нужной строки с тегами всех строк кэш-памяти, пока не обнаружит совпадение.

Другое решение — отображать страницы полностью, но при этом не требовать присутствия всех строк кэш-памяти. В этом случае аппаратному обеспечению понадобится битовое отображение для каждой страницы, где один бит для каждой строки указывает на присутствие или отсутствие этой строки. Если строка присутствует, она должна находиться в правильной позиции на этой странице. Если она отсутствует, то любая попытка использовать ее вызовет прерывание, что позволит программному обеспечению найти нужную строку и ввести ее.

Таким образом, система будет искать только те строки, которые действительно находятся в удаленной памяти. Одно из решений — предоставить каждой странице собственную машину (ту, которая содержит элемент каталога данной страницы, а не ту, в которой находятся данные). Затем можно отправить сообщение в собственную машину, чтобы найти местоположение данной строки. Другое решение — организовать память в виде дерева и осуществлять поиск по направлению вверх, пока не будет обнаружена требующаяся строка.

Вторая проблема связана с удалением последней копии. Как и в машине CC-NUMA, строка кэш-памяти может находиться одновременно в нескольких узлах. Если происходит промах кэша, строку нужно откуда-то вызвать, а это значит, что ее нужно отбросить. А что произойдет, если выбранная строка окажется последней копией? В этом случае ее нельзя отбрасывать.

Одно из возможных решений — вернуться к каталогу и проверить, существуют ли другие копии. Если да, то строку можно смело выбрасывать. Если нет, то ее нужно переместить куда-либо еще. Другое решение — пометить одну из копий каждой строки кэш-памяти как главную копию и никогда ее не выбрасывать. При таком подходе не требуется проверка каталога. Машина COMA обещает обеспечить лучшую производительность, чем CC-NUMA, но дело в том, что было построено очень мало машин COMA, и нужно больше опыта. До настоящего момента создано всего две машины COMA: KSR-1 [20] и Data Diffusion Machine [53]. Дополнительную информацию о машинах COMA можно найти в книгах [36, 67, 98, 123].

Мультикомпьютеры с передачей сообщений

Как видно из схемы на рис. 8.12, существует два типа параллельных процессоров MIMD: мультипроцессоры и мультикомпьютеры. В предыдущем разделе мы рассматривали мультипроцессоры. Мы увидели, что мультипроцессоры могут иметь разделенную память, доступ к которой можно получить с помощью обычных команд LOAD и STORE. Такая память реализуется разными способами, включая отслеживающие шины, многоступенчатые сети, а также различные схемы на основе каталога. Программы, написанные для мультипроцессора, могут получать доступ к любому месту в памяти, не имея никакой информации о внутренней топологии или схеме реализации. Именно благодаря такой иллюзии мультипроцессоры весьма популярны.

Однако мультипроцессоры имеют и некоторые недостатки, поэтому мультикомпьютеры тоже очень важны. Во-первых, мультипроцессоры нельзя расширить до больших размеров. Чтобы расширить машину Interprise 10000 до 64 процессоров, пришлось добавить огромное количество аппаратного обеспечения.

В Sequent NUMA-Q дошли до 256 процессоров, но ценой неодинакового времени доступа к памяти. Ниже мы рассмотрим два мультикомпьютера, которые содержат 2048 и 9152 процессора соответственно. Через много лет кто-нибудь сконструирует мультипроцессор, содержащий 9000 узлов, но к тому времени мультикомпьютеры будут содержать уже 100 000 узлов.

Кроме того, конфликтная ситуация при обращении к памяти в мультипроцессоре может сильно повлиять на производительность. Если 100 процессоров постоянно пытаются считывать и записывать одни и те же переменные, конфликтная ситуация для различных модулей памяти, шин и каталогов может сильно ударить по производительности.

Вследствие этих и других факторов разработчики проявляют огромный интерес к параллельным компьютерам, в которых каждый процессор имеет свою собственную память, к которой другие процессоры не могут получить прямой доступ. Это мультикомпьютеры. Программы на разных процессорах в мультикомпьютере взаимодействуют друг с другом с помощью примитивов send и receive, которые используются для передачи сообщений (поскольку они не могут получить доступ к памяти других процессоров с помощью команд LOAD и STORE). Это различие полностью меняет модель программирования.

Каждый узел в мультикомпьютере состоит из одного или нескольких процессоров, ОЗУ (общее для процессоров только данного узла), диска и(или) других устройств ввода-вывода, а также процессора передачи данных. Процессоры передачи данных связаны между собой по высокоскоростной коммуникационной сети (см. раздел «Сети межсоединений»). Используется множество различных топологий, схем коммутации и алгоритмов выбора маршрута. Все мультикомпьютеры сходны в одном: когда программа выполняет примитив send, процессор передачи данных получает уведомление и передает блок данных в целевую машину (возможно, после предварительного запроса и получения разрешения). Схема мультикомпьютера показана на рис. 8.28.

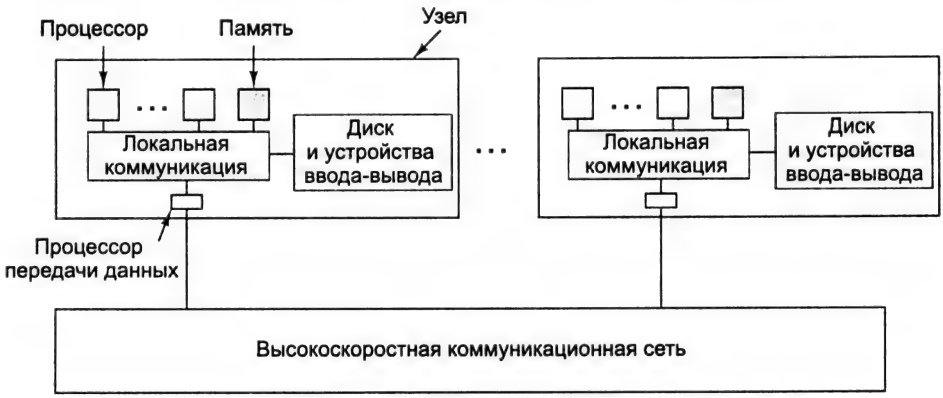


Рис. 8.28. Схема мультикомпьютера

Мультикомпьютеры бывают разных типов и размеров, поэтому очень трудно привести хорошую классификацию. Тем не менее можно назвать два общих типа: MPP и COW. Ниже мы рассмотрим каждый из этих типов.

MPP — процессоры с массовым параллелизмом

MPP (Massively Parallel Processors — процессоры с массовым параллелизмом) — это огромные суперкомпьютеры стоимостью несколько миллионов долларов. Они используются в различных науках и промышленности для выполнения сложных вычислений, для обработки большого числа транзакций в секунду или для хранения больших баз данных и управления ими.

В большинстве таких машин используются стандартные процессоры. Это могут быть процессоры Intel Pentium, Sun UltraSPARC, IBM RS/6000 и DEC Alpha. Отличает мультикомпьютеры то, что в них используется сеть, по которой можно передавать сообщения, с низким временем ожидания и высокой пропускной способностью. Обе характеристики очень важны, поскольку большинство сообщений малы по размеру (менее 256 байтов), но при этом суммарная нагрузка в большей степени зависит от длинных сообщений (более 8 Кбайт).

Еще одна характеристика MPP — огромная производительность процесса ввода-вывода. Часто приходится обрабатывать огромные массивы данных, иногда терабайты. Эти данные должны быть распределены по дискам, и их нужно перемещать в машине с большой скоростью.

Следующая специфическая черта MPP — отказоустойчивость. При наличии не одной тысячи процессоров несколько неисправностей в неделю неизбежны. Прекращать работу системы из-за неполадок в одном из процессоров было бы неприемлемо, особенно если такая неисправность ожидается каждую неделю. Поэтому большие MPP всегда содержат специальное аппаратное и программное обеспечение для контроля системы, обнаружения неполадок и их исправления.

Было бы неплохо теперь рассмотреть основные принципы разработки MPP, но, по правде говоря, их не так много. MPP представляет собой набор более или

менее стандартных узлов, которые связаны друг с другом высокоскоростной сетью. Поэтому ниже мы просто рассмотрим несколько конкретных примеров систем MPP: Cray T3E и Intel/Sandia Option Red.

Cray T3E

В семейство T3E (последователя T3D) входят самые последние суперкомпьютеры, восходящие к компьютеру 6600. Различные модели — T3E, T3E-900 и T3E-1200 — идентичны с точки зрения архитектуры и различаются только ценой и производительностью (например, 600, 900 или 1200 мегафлопов на процессор). Мегафлоп — это 1 млн операций с плавающей точкой/с. (FLOP — FLoating-point OPerations — операции с плавающей точкой). В отличие от 6600 и Cray-1, в которых очень мало параллелизма, эти машины могут содержать до 2048 процессоров. Мы используем термин T3E для обозначения всего семейства, но величины производительности будут приведены для машины T3E-1200. Эти машины продает компания Cray Research, филиал Silicon Graphics. Они применяются для разработки лекарственных препаратов, поиска нефти и многих других задач.

В системе T3E используются процессоры DEC Alpha 21164. Это суперскалярный процессор RISC, способный выдавать 4 команды за цикл. Он работает с частотой 300, 450 и 600 МГц в зависимости от модели. Тактовая частота — основное различие между разными моделями T3E. Alpha — это 64-битная машина с 64-битными регистрами. Размер виртуальных адресов ограничен до 43 битов, а физических — до 40 битов. Таким образом, возможен доступ к 1 Тбайт физической памяти.

Каждый процессор Alpha имеет двухуровневую кэш-память, встроенную в микросхему процессора. Кэш-память первого уровня содержит 8 Кбайт для команд и 8 Кбайт для данных. Кэш-память второго уровня — это смежная трехвходовая ассоциативная кэш-память на 96 Кбайт, содержащая и команды и данные вместе. Кэш-память обоих уровней содержит команды и данные только из локального ОЗУ, а это может быть до 2 Гбайт на процессор. Поскольку максимальное число процессоров равно 2048, общий объем памяти может составлять 4 Тбайт.

Каждый процессор Alpha заключен в особую схему, которая называется **оболочкой** (shell) (рис. 8.29). Оболочка содержит память, процессор передачи данных и 512 специальных регистров (так называемых **E-регистров**). Эти регистры могут загружаться адресами удаленной памяти с целью чтения или записи слов из удаленной памяти (или блоков из 8 слов). Это значит, что в машине T3E есть доступ к удаленной памяти, но осуществляется он не с помощью обычных команд `LOAD` и `STORE`. Эта машина представляет собой гибрид между NC-NUMA и MPP, но все-таки больше похожа на MPP. Непротиворечивость памяти гарантируется, поскольку слова, считываемые из удаленной памяти, не попадают в кэш-память.

Узлы в машине T3E связаны двумя разными способами (см. рис. 8.29). Основная топология — дуплексный 3-мерный тор. Например, система, содержащая 512 узлов, может быть реализована в виде куба 8×8×8. Каждый узел в 3-мерном торе имеет 6 каналов связи с соседними узлами (по направлению вперед, назад, вправо, влево, вверх и вниз). Скорость передачи данных в этих каналах связи равна 480 Мбайт/с в любом направлении.

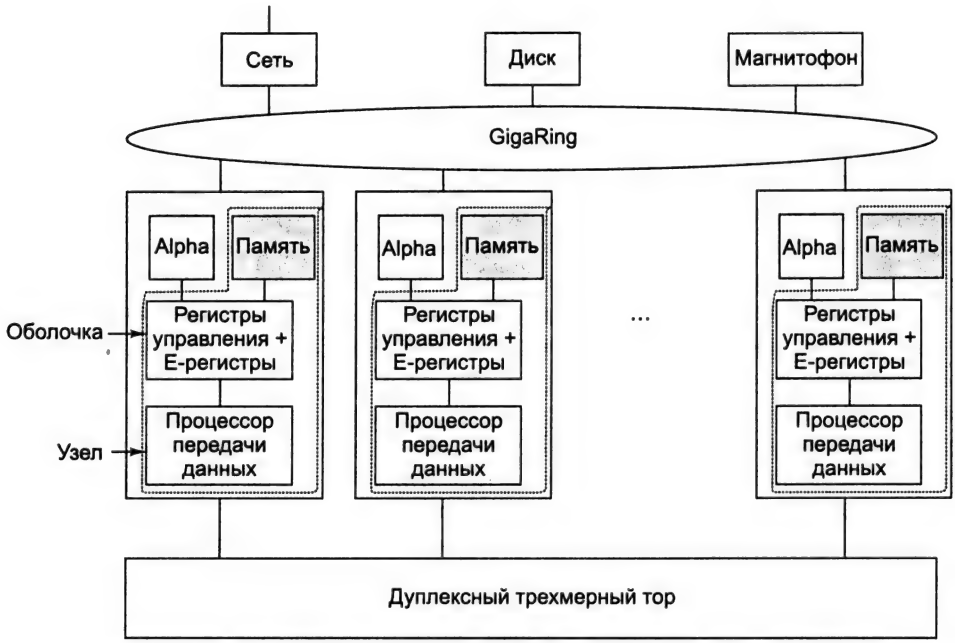


Рис. 8.29. Cray Research T3E

Узлы также связаны одним или несколькими **GigaRings** — подсистемами ввода-вывода с коммутацией пакетов, обладающими высокой пропускной способностью. Узлы используют эту подсистему для взаимодействия друг с другом, а также с сетями, дисками и другими периферическими устройствами. Они по ней посылают пакеты размером до 256 байтов. Каждый GigaRing состоит из пары колец шириной в 32 бита, которые соединяют узлы процессоров со специальными узлами ввода-вывода. Узлы ввода-вывода содержат гнезда для сетевых карт (например, HIPPI, Ethernet, ATM, FDDI), дисков и других устройств.

В системе T3E может быть до 2048 узлов, поэтому неисправности будут происходить регулярно. По этой причине в системе на каждые 128 пользовательских узлов содержится один запасной узел. Испорченные узлы могут быть замещены запасными во время работы системы без перезагрузки. Кроме пользовательских и запасных узлов есть узлы, которые предназначены для запуска серверов операционной системы, поскольку пользовательские узлы запускают не всю систему, а только ядро. В данном случае используется операционная система UNIX.

Intel/Sandia Option Red

Компьютеры с высокой производительностью и вооруженные силы идут в США рука об руку с 1943 года, начиная с машины ENIAC, первого электронного компьютера. Связь между американскими вооруженными силами и высокоскоростными вычислениями до сих пор продолжается. В середине 90-х годов департаменты обороны и энергетики приступили к выполнению программы разработки 5 систем MPP, которые будут работать со скоростью 1, 3, 10, 30 и 100 терафлопов/с соот-

ветственно. Для сравнения: 100 терафлопов (10^{14} операций с плавающей точкой в секунду) — это в 500000 раз больше, чем мощность процессора Pentium Pro, работающего с частотой 200 МГц.

В отличие от машины T3E, которую можно купить в магазине (правда, за большие деньги), машины, работающие со скоростью 10^{14} операций с плавающей точкой, — это уникальные системы, распределяемые в конкурентных торгах Департаментом энергетики, который руководит национальными лабораториями. Компания Intel выиграла первый контракт; IBM выиграла следующие два. Если вы планируете вступить в соревнование в будущем, вам понадобится 80 млн долларов. Эти машины предназначены для военных целей. Какой-то сообразительный работник Пентагона придумал патриотические названия для первых трех машин: red, white и blue (красный, белый и синий — цвета флага США). Первая машина, выполнявшая 10^{14} операций с плавающей точкой, называлась **Option Red** (Sandia National Laborotary, декабрь 1996), вторая — **Option Blue** (1999), а третья — **Option White** (2000). Ниже мы будем рассматривать первую из этих машин, Option Red.

Машина Option Red состоит из 4608 узлов, которые организованы в трехмерную сетку. Процессоры запакованы на платах двух разных типов. Платы **kestrel** используются в качестве вычислительных узлов, а платы **eagle** используются для сервисных, дисковых, сетевых узлов и узлов загрузки. Машина содержит 4536 вычислительных узлов, 32 сервисных узла, 32 дисковых узла, 6 сетевых узлов и 2 узла загрузки.

Плата kestrel (рис. 8.30, *а*) содержит 2 логических узла, каждый из которых включает 2 процессора Pentium Pro на 200 МГц и разделенное ОЗУ на 64 Мбайт. Каждый узел kestrel содержит собственную 64-битную локальную шину и собственную микросхему **NIC (Network Interface Chip — сетевой адаптер)**. Две микросхемы NIC связаны вместе, поэтому только одна из них подсоединена к сети, что делает систему более компактной. Платы eagle также содержат процессоры Pentium Pro, но всего два на каждую плату. Кроме того, они отличаются высокой производительностью процесса ввода-вывода.

Платы связаны в виде решетки 32×38×2 в виде двух взаимосвязанных плоскостей 32×38 (размер решетки продиктован целями компоновки, поэтому не во всех узлах решетки находятся платы). В каждом узле находится маршрутизатор с шестью каналами связи: вперед, назад, вправо, влево, с другой плоскостью и с платой kerstel или eagle. Каждый канал связи может передавать информацию одновременно в обоих направлениях со скоростью 400 Мбайт/с. Применяется маршрутизация «червоточина», чтобы сократить время ожидания.

Применяется пространственная маршрутизация, когда пакеты сначала потенциально перемещаются в другую плоскость, затем вправо-влево, затем вперед-назад и, наконец, в нужную плоскость, если они еще не оказались в нужной плоскости. Два перемещения между плоскостями нужны для повышения отказоустойчивости. Предположим, что пакет нужно переместить в соседний узел, находящийся впереди исходного, но канал связи между ними поврежден. Тогда сообщение можно отправить в другую плоскость, затем на один узел вперед, а затем обратно в первую плоскость, минуя таким образом поврежденный канал связи.

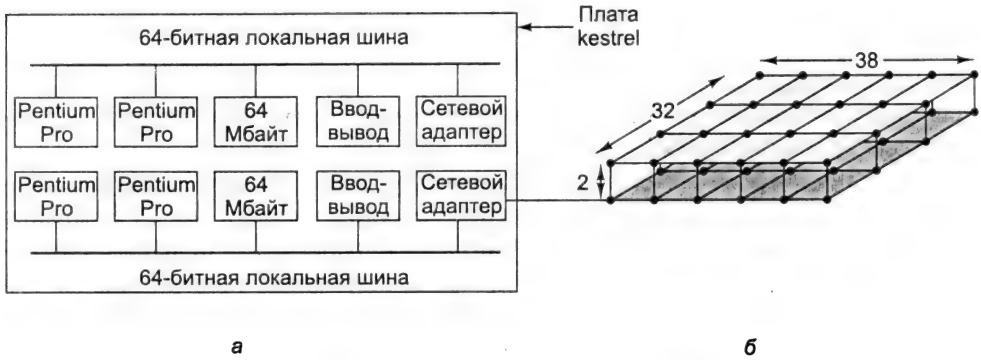


Рис. 8.30. Система Intel/Sandia Option Red: плата kestrel (*а*); сеть (*б*)

Систему можно логически разделить на 4 части: сервис, вычисление, ввод-вывод и система. Сервисные узлы — это машины UNIX общего назначения с разделением времени, которые позволяют программистам писать и отлаживать свои программы. Вычислительные узлы запускают большие приложения. Они запускают не всю систему UNIX, а только микроядро, которое называется **кугуаром** (coguar)[1]. Узлы ввода-вывода управляют 640 дисками, содержащими более 1 Тбайт данных. Есть два независимых набора узлов ввода-вывода. Узлы первого типа предназначены для секретной военной работы, а узлы второго типа — для несекретной работы. Эти два набора вводятся и удаляются из системы вручную, поэтому в каждый момент времени подсоединен только один набор узлов, чтобы предотвратить утечку информации с секретных дисков на несекретные диски. Наконец, системные узлы используются для загрузки системы.

COW — Clusters of Workstations (кластеры рабочих станций)

Второй тип мультикомпьютеров — это системы **COW (Cluster of Workstations — кластер рабочих станций)** или **NOW (Network of Workstations — сеть рабочих станций)** [8, 90]. Обычно он состоит из нескольких сотен персональных компьютеров или рабочих станций, соединенных посредством сетевых плат. Различие между MPP и COW аналогично разнице между большой вычислительной машиной и персональным компьютером. У обоих есть процессор, ОЗУ, диски, операционная система и т. д. Но в большой вычислительной машине все это работает гораздо быстрее (за исключением, может быть, операционной системы). Однако они применяются и управляются по-разному. Это же различие справедливо для MPP и COW.

Процессоры в MPP — это обычные процессоры, которые любой человек может купить. В системе T3E используются процессоры Alpha; в системе Option Red — процессоры Pentium Pro. Ничего специфического. Используются обычные динамические ОЗУ и система UNIX.

[1] Кугуар (или пума), как известно, очень быстрое животное. Видимо, этот термин был использован с целью отметить высокое быстродействие микроядра. — *Примеч. научн. ред.*

Исторически система MPP отличалась высокоскоростной сетью. Но с появлением коммерческих высокоскоростных сетей это отличие начало сглаживаться. Например, исследовательская группа автора данной книги собрала систему COW, которая называется **DAS (Distributed ASCII Supercomputer).** Она состоит из 128 узлов, каждый из которых содержит процессор Pentium Pro на 200 МГц и ОЗУ на 128 Мбайт (см. http://www.cs.vu.nl/~bal/das.html). Узлы организованы в 2-мерный тор. Каналы связи могут передавать информацию со скоростью 160 Мбайт/с в обоих направлениях одновременно. Эти характеристики практически не отличаются от характеристик машины Option Red: скорость передачи информации по каналам связи в два раза ниже, но размер ОЗУ каждого узла в два раза больше. Единственное существенное различие состоит в том, что бюджет Sandia был значительно больше. Технически эти две системы практически не различаются.

Преимущество системы COW над MPP в том, что COW полностью состоит из доступных компонентов, которые можно купить. Эти части выпускаются большими партиями. Эти части, кроме того, существуют на рынке с жесткой конкуренцией, из-за которой производительность растет, а цены падают. Вероятно, системы COW постепенно вытеснят MMP, подобно тому как персональные компьютеры вытеснили большие вычислительные машины, которые применяются теперь только в специализированных областях.

Существует множество различных видов COW, но доминируют два из них: централизованные и децентрализованные. Централизованные системы COW представляют собой кластер рабочих станций или персональных компьютеров, смонтированных в большой блок в одной комнате. Иногда они компонуются более компактно, чем обычно, чтобы сократить физические размеры и длину кабеля. Как правило, эти машины гомогенны и не имеют никаких периферических устройств, кроме сетевых карт и, возможно, дисков. Гордон Белл (Gordon Bell), разработчик PDP-11 и VAX, назвал такие машины **«автономными рабочими станциями»** (поскольку у них не было владельцев).

Децентрализованная система COW состоит из рабочих станций или персональных компьютеров, которые раскиданы по зданию или по территории учреждения. Большинство из них простаивают много часов в день, особенно ночью. Обычно они связаны через локальную сеть. Они гетерогенны и имеют полный набор периферийных устройств. Самое важное, что многие компьютеры имеют своих владельцев.

Планирование

Возникает вопрос: чем отличается децентрализованная система COW от локальной сети, соединяющей пользовательские машины? Отличие связано с программным обеспечением и не имеет никакого отношения к аппаратному обеспечению. В локальной сети пользователи работают с персональными машинами и используют только их для своей работы. Децентрализованная система COW, напротив, является общим ресурсом, которому пользователи могут поручить работу, требующую для выполнения нескольких процессоров. Чтобы система COW могла обрабатывать запросы от нескольких пользователей, каждому из которых нужно несколько процессоров, этой системе необходим планировщик заданий.

Рассмотрим самую простую модель планирования. Должно быть известно, сколько процессоров нужно для каждой работы (задачи). Тогда задачи выстраиваются в порядке FIFO («первым вошел — первым вышел») (рис. 8.31, *а*). Когда первая задача начала выполняться, происходит проверка, есть ли достаточное количество процессоров для выполнения задачи, следующей по очереди. Если да, то она тоже начинает выполняться и т. д. Если нет, то система ждет, пока не появится достаточное количество процессоров. В нашем примере система COW содержит 8 процессоров, но она вполне могла бы содержать 128 процессоров, расположенных в блоках по 16 процессоров (получилось бы 8 групп процессоров) или в какой-нибудь другой комбинации.

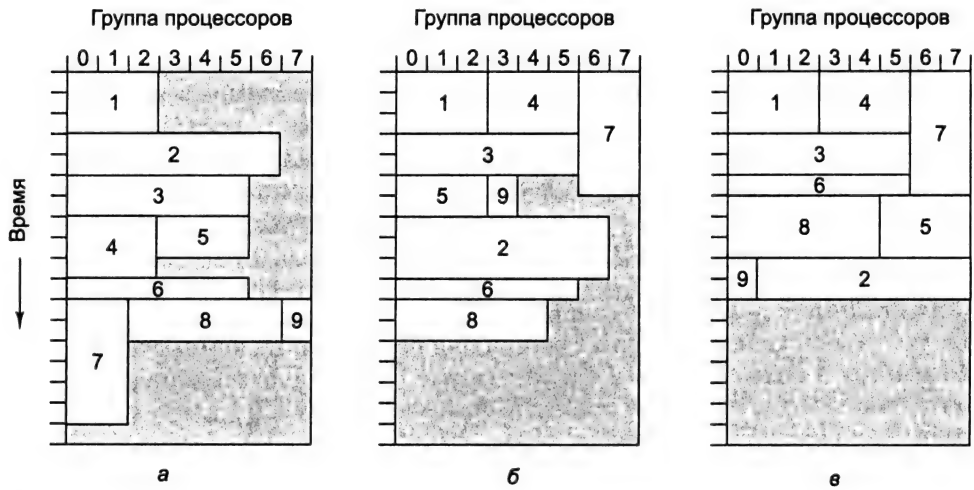


Рис. 8.31. Планирование работы в системе COW: FIFO («первым вошел — первым вышел») (*а*); без блокировки начала очереди (*б*); заполнение прямоугольника «процессоры-время» (*в*). Серым цветом показаны свободные процессоры

В глубже разработанном алгоритме задачи, которые не соответствуют количеству имеющихся в наличии процессоров, пропускаются и берется первая задача, для которой процессоров достаточно. Всякий раз, когда завершается выполнение задачи, очередь из оставшихся задач проверяется в порядке «первым вошел — первым вышел». Результат применения этого алгоритма изображен на рис. 8.31, *б*.

Еще более сложный алгоритм требует, чтобы было известно, сколько процессоров нужно для каждой задачи и сколько минут займет ее выполнение. Располагая такой информацией, планировщик заданий может попытаться заполнить прямоугольник «процессоры—время». Это особенно эффективно, когда задачи представлены на рассмотрение днем, а выполняться будут ночью. В этом случае планировщик заданий получает всю информацию о задачах заранее и может выполнять их в оптимальном порядке, как показано на рис. 8.31, *в*.

Коммерческие сети межсоединений

В этом разделе мы рассмотрим некоторые технологии связи. Наш первый пример — система Ethernet. Существует три версии этой системы: classic Ethernet, fast

Ethernet и gigabit Ethernet. Они работают со скоростью 10, 100 и 1000 Мбит/с (1,25, 12,5 и 125 Мбайт/с)[1] соответственно. Все они совместимы относительно среды, формата пакетов и протоколов[2]. Отличие только в производительности.

Каждый компьютер в сети Ethernet содержит микросхему Ethernet, обычно на съемной плате. Изначально провод из платы вводился в середину толстого медного кабеля, это называлось **«зуб вампира»**. Позднее появились более тонкие кабели и Т-образные коннекторы. В любом случае платы Ethernet на всех машинах соединены электрически, как будто они соединены пайкой. Схема подсоединения трех машин к сети Ethernet изображена на рис. 8.32, *а*.

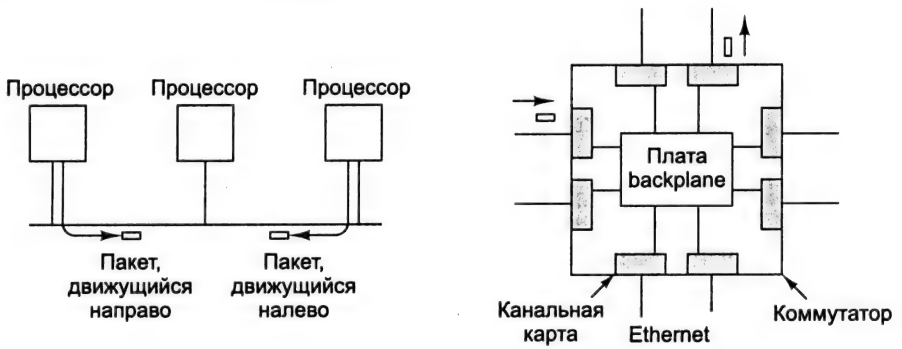


Рис. 8.32. Три компьютера в сети Ethernet (*а*); коммутатор Ethernet (*б*)

В соответствии с протоколом Ethernet, если машине нужно послать пакет, сначала она должна проверить, не совершает ли передачу в данный момент какая-либо другая машина. Если кабель свободен, то машина просто посылает пакет. Если кабель занят, то машина ждет окончания передачи и только после этого посылает пакет. Если две машины начинают передачу пакета одновременно, происходит конфликтная ситуация. Обе машины определяют, что произошла конфликтная ситуация, останавливают передачу, затем останавливаются на произвольный период времени и пробуют снова. Если конфликтная ситуация случается во второй раз, они снова останавливаются и снова начинают передачу пакетов, удваивая среднее время ожидания с каждой последующей конфликтной ситуацией.

Дело в том, что «зубы вампира» легко ломаются, а определить неполадку в кабеле очень трудно. По этой причине появилась новая разработка, в которой кабель из каждой машины подсоединяется к **сетевому концентратору (хабу)**. По суще-

[1] Соотнесение автором скоростных показателей упоминаемых технологий, выраженных отношением скорости передачи бит/с, с отношениями Мбайт/с неправомочно. Ни одна их этих технологий не позволяет передать по сети соответствующее количество байтов за секунду. Даже теоретически возможная скорость для стандарта Ethernet лежит в интервале 800–850 Кбайт/с. Дело в том, что для передачи информации необходимо использовать служебные коды, к тому же передача осуществляется кадрами (фреймами) относительно небольшой длины (1500 байтов), после чего сетевой адаптер обязательно должен освободить среду передачи на фиксированный промежуток времени (с тем, чтобы другие сетевые адаптеры тоже могли воспользоваться средой передачи). — *Примеч. научн. ред.*

[2] Это высказывание является чересчур смелым. В действительности совместимость имеет очень много ограничений, и общим у них следует считать метод доступа к среде передачи. — *Примеч. научн. ред.*

ству, это то же самое, что и в первой разработке, но производить ремонт здесь проще, поскольку кабели можно отсоединять от сетевого концентратора по очереди, пока поврежденный кабель не будет изолирован.

Третья разработка — **Ethernet с использованием коммутаторов** — показана на рис. 8.32, *б*. Здесь сетевой концентратор заменен устройством, содержащим высокоскоростную плату backplane, к которой можно подсоединять **канальные карты**. Каждая канальная карта принимает одну или несколько сетей Ethernet, и разные карты могут воспринимать разные скорости, поэтому classic, fast и gigabit Ethernet могут быть связаны вместе.

Когда пакет поступает в канальную карту, он временно сохраняется там в буфере, пока канальная карта не отправит запрос и не получит доступ к плате backplane, которая функционирует почти как шина. Если пакет был перемещен в канальную карту, к которой подсоединена целевая машина, он может направляться к этой машине. Если каждая канальная карта содержит только один Ethernet и этот Ethernet имеет только одну машину, конфликтных ситуаций больше не возникнет, хотя пакет может быть потерян из-за переполнения буфера в канальной карте. Gigabit Ethernet с использованием коммутаторов с одной машиной на Ethernet и высокоскоростной платой backplane имеет потенциальную производительность (по крайней мере, это касается пропускной способности) в 4 раза меньше, чем каналы связи в машине T3E, но стоит значительно дешевле.

Но при большом количестве канальных карт обычная плата backplane не сможет справляться с такой нагрузкой, поэтому необходимо подсоединить несколько машин к каждой сети Ethernet, вследствие чего опять возникнут конфликтные ситуации. Однако с точки зрения соотношения цены и производительности сеть на основе gigabit Ethernet с использованием коммутаторов — серьезный конкурент на компьютерном рынке.

Следующая технология связи, которую мы рассмотрим, — это **ATM (Asynchronous Transfer Mode — асинхронный режим передачи)**. Технология ATM была разработана международным консорциумом телефонных компаний в качестве замены существующей телефонной системы на новую, полностью цифровую. Основная идея проекта состояла в том, чтобы каждый телефон и каждый компьютер в мире связать с помощью безошибочного цифрового битового канала со скоростью передачи данных 155 Мбит/с (позднее 622 Мбит/с). Но осуществить это на практике оказалось не так просто. Тем не менее многие компании сейчас выпускают съемные платы для персональных компьютеров со скоростью передачи данных 155 Мбит/с или 622 Мбит/с. Вторая скорость, **OC-12**, хорошо подходит для мультикомпьютеров.

Провод или стекловолокно, отходящее от платы ATM, переходит в переключатель ATM — устройство, похожее на коммутатор Ethernet. В него тоже поступают пакеты и сохраняются в буфере в канальных картах, а затем поступают в исходящую канальную карту для передачи в пункт назначения. Однако у Ethernet и ATM есть существенные различия.

Во-первых, поскольку ATM была разработана для замещения телефонной системы, она представляет собой сеть с маршрутизацией информации. Перед отправкой пакета в пункт назначения исходная машина должна установить виртуальную цепь от исходного пункта через один или несколько коммутаторов ATM в конеч-

ный пункт. На рис. 8.33. показаны две виртуальные цепи. В сети Ethernet, напротив, нет никаких виртуальных цепей. Поскольку установка виртуальной цепи занимает некоторое количество времени, каждая машина в мультикомпьютере должна устанавливать виртуальную цепь со всеми другими машинами при запуске и использовать их при работе. Пакеты, отправленные по виртуальной цепи, всегда будут доставлены в правильном порядке, но буферы канальных карт могут переполняться, как и в сети Ethernet с коммутаторами, поэтому доставка не гарантируется.

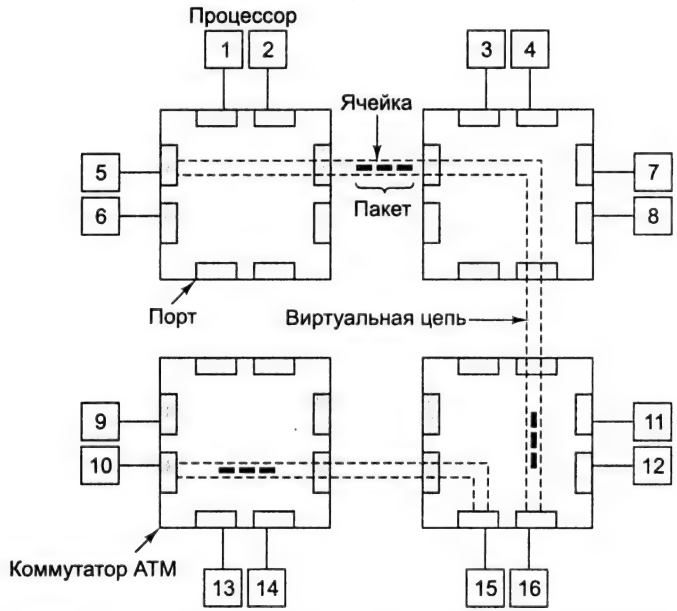


Рис. 8.33. 16 процессоров, связанных четырьмя переключателями ATM. Пунктиром показаны две виртуальные цепи (канала)

Во-вторых, Ethernet может передавать целые пакеты (до 1500 байтов данных) одним блоком. В ATM все пакеты разбиваются на ячейки по 53 байта. Пять из этих байтов — это поля заголовка, которые сообщают, какой виртуальной цепи принадлежит ячейка, что это за ячейка, каков ее приоритет, а также некоторые другие сведения. Полезная нагрузка составляет 48 байтов. Разбиение пакетов на ячейки и их компоновку в конце пути совершает аппаратное обеспечение.

Наш третий пример — сеть Myrinet — съемная плата, которая производится одной калифорнийской компанией и пользуется популярностью у разработчиков систем COW [18]. Здесь используется та же модель, что и в Ethernet и ATM, где каждая съемная плата подсоединяется к коммутатору, а коммутаторы могут соединяться в любой топологии. Каналы связи сети Myrinet дуплексные, они передают информацию со скоростью 1,28 Гбит/с в обоих направлениях. Размер пакетов неограничен, а каждый коммутатор представляет собой полное пересечение, что дает малое время ожидания и высокую пропускную способность.

Myrinet пользуется популярностью у разработчиков систем COW, поскольку платы в этой сети содержат программируемый процессор и большое ОЗУ. Хотя

Myrinet появилась со своей стандартной операционной системой, многие исследовательские группы уже разработали свои собственные операционные системы. У них появились дополнительные функции и повысилась производительность (см., например, [17, 107, 155]). Из типичных особенностей можно назвать защиту, управление потоком, надежное широковещание и мультивещание, а также возможность запускать часть кода прикладной программы на плате.

Связное программное обеспечение для мультикомпьютеров

Для программирования мультикомпьютера требуется специальное программное обеспечение (обычно это библиотеки), чтобы обеспечить связь между процессами и синхронизацию. В этом разделе мы расскажем о таком программном обеспечении. Отметим, что большинство этих программных пакетов работают в системах MPP и COW.

В системах с передачей сообщений два и более процессов работают независимо друг от друга. Например, один из процессов может производить какие-либо данные, а другой или несколько других процессов могут потреблять их. Если у отправителя есть еще данные, нет никакой гарантии, что получатель (получатели) готов принять эти данные, поскольку каждый процесс запускает свою программу.

В большинстве систем с передачей сообщений имеется два примитива `send` и `receive`, но возможны и другие типы семантики. Ниже даны три основных варианта:

1. Синхронная передача сообщений.
2. Буферная передача сообщений.
3. Неблокируемая передача сообщений.

Синхронная передача сообщений. Если отправитель выполняет операцию `send`, а получатель еще не выполнил операцию `receive`, то отправитель блокируется до тех пор, пока получатель не выполнит операцию `receive`, а в это время сообщение копируется. Когда к отправителю возвращается управление, он уже знает, что сообщение было отправлено и получено. Этот метод имеет простую семантику и не требует буферизации. Но у него есть большой недостаток: отправитель блокируется до тех пор, пока получатель не примет и не подтвердит прием сообщения.

Буферная передача сообщений. Если сообщение отправляется до того, как получатель готов его принять, это сообщение временно сохраняется где-либо, например в почтовом ящике, и хранится там, пока получатель не возьмет его оттуда. При таком подходе отправитель может продолжать работу после операции `send`, даже если получатель в этот момент занят. Поскольку сообщение уже отправлено, отправитель может снова использовать буфер сообщений сразу же. Такая схема сокращает время ожидания. Вообще говоря, как только система отправила сообщение, отправитель может продолжать работу. Однако нет никаких гарантий, что сообщение было получено. Даже при надежной системе коммуникаций получатель мог сломаться еще до получения сообщения.

Неблокируемая передача сообщений. Отправитель может продолжать работу сразу после вызова. Библиотека только сообщает операционной системе, что

она сделает вызов позднее, когда у нее будет время. В результате отправитель вообще не блокируется. Недостаток этого метода состоит в том, что когда отправитель продолжает работу после совершения операции send, он не может снова использовать буфер сообщений, так как есть вероятность, что сообщение еще не отправлено. Отправитель каким-то образом должен определять, когда он может снова использовать буфер. Например, можно опрашивать систему или совершать прерывание, когда буфер имеется в наличии. В обоих случаях программное обеспечение очень сложное.

В следующих двух разделах мы рассмотрим две популярные системы с передачей сообщений, которые применяются во многих мультикомпьютерах: PVM и MPI. Существуют и другие системы, но эти две наиболее распространенные.

PVM — виртуальная машина параллельного действия

PVM (Parallel Virtual Machine — виртуальная машина параллельного действия) — это система с передачей сообщений, изначально разработанная для машин COW с операционной системой UNIX [45, 142]. Позднее она стала применяться в других машинах, в том числе в системах MPP. Это самодостаточная система с управлением процессами и системой ввода-вывода.

PVM состоит из двух частей: библиотеки, вызываемой пользователем, и «сторожевого» процесса, который работает постоянно на каждой машине в мультикомпьютере. Когда PVM начинает работу, она определяет, какие машины должны быть частью ее виртуального мультикомпьютера. Для этого она читает конфигурационный файл. «Сторожевой» процесс запускается на каждой из этих машин. Машины можно добавлять и убирать, вводя команды на консоли PVM.

Можно запустить n параллельных процессов с помощью команды

```
spawn -count n prog
```

Каждый процесс запустит *prog*. PVM решает, куда поместить процессы, но пользователь может сам подменять их с помощью аргументов команды spawn. Процессы могут запускаться из работающего процесса — для этого нужно вызвать процедуру *Pvm_spawn*. Процессы могут быть организованы в группы, причем состав групп может меняться во время выполнения программы.

Взаимодействие в машине PVM осуществляется с помощью примитивов для передачи сообщений таким образом, чтобы взаимодействовать могли машины с разными системами счисления. Каждый процесс PVM в каждый момент времени имеет один активный пересылочный буфер и один активный приемный буфер. Отправляя сообщение, процесс вызывает библиотечные процедуры, запаковывающие значения с самоописанием в активный пересылочный буфер, чтобы получатель мог узнать их и преобразовать в исходный формат.

Когда сообщение скомпоновано, отправитель вызывает библиотечную процедуру *pvm_send*, которая представляет собой блокирующий сигнал send. Получатель может поступить по-разному. Во-первых, он может вызвать процедуру *pvm_recv*, которая блокирует получателя до тех пор, пока не придет подходящее сообщение. Когда вызов возвратится, сообщение будет в активном приемном буфере. Оно может быть распаковано и преобразовано в подходящий для данной машины формат с помощью набора распаковывающих процедур. Во-вторых, получатель может вызвать процедуру *pvm_trecv*, которая блокирует получателя на

определенный промежуток времени, и если подходящего сообщения за это время не пришло, он разблокируется. Процедура нужна для того, чтобы не заблокировать процесс навсегда. Третий вариант — процедура *pvm_nrecv*, которая сразу же возвращает значение — это может быть либо сообщение, либо указание на отсутствие сообщений. Вызов можно повторять, чтобы опрашивать входящие сообщения.

Помимо всех этих примитивов PVM поддерживает широковещание (процедура *pvm_bcast*) и мультивещание (процедура *pvm_mcast*). Первая процедура отправляет сообщение всем процессам в группе, вторая посылает сообщение только некоторым процессам, входящим в определенный список.

Синхронизация между процессами осуществляется с помощью процедуры *pvm_barrier*. Когда процесс вызывает эту процедуру, он блокируется до тех пор, пока определенное число других процессов не достигнет барьера и они не вызовут эту же процедуру. Существуют другие процедуры для управления главной вычислительной машиной, группами, буферами, для передачи сигналов, проверки состояния и т. д. PVM — это простой, легкий в применении пакет, имеющийся в наличии в большинстве компьютеров параллельного действия, что и объясняет его популярность.

MPI — интерфейс с передачей сообщений

Следующий пакет для программирования мультикомпьютеров — **MPI (Message-Passing Interface — интерфейс с передачей сообщений)**. MPI гораздо сложнее, чем PVM. Он содержит намного больше библиотечных вызовов и намного больше параметров на каждый вызов. Первая версия MPI, которая сейчас называется MPI-1, была дополнена второй версией, MPI-2, в 1997 году. Ниже мы в двух словах расскажем о MPI-1, а затем посмотрим, что нового появилось в MPI-2. Более подробную информацию об MPI см. в книгах [52, 134].

MPI-1, в отличие от PVM, никак не связана с созданием процессов и управлением процессами. Создавать процессы должен сам пользователь с помощью локальных системных вызовов. После создания процессы организуются в группы, которые уже не изменяются. Именно с этими группами и работает MPI.

В основе MPI лежат 4 понятия: коммуникаторы, типы передаваемых данных, операции коммуникации и виртуальные топологии. **Коммуникатор** — это группа процессов плюс контекст. Контекст — это метка, которая идентифицирует что-либо (например, фазу выполнения). В процессе отправки и получения сообщений контекст может использоваться для того, чтобы несвязанные сообщения не мешали друг другу.

Сообщения могут быть разных типов: символьные, целочисленные (short, regular и long integers), с обычной и удвоенной точностью, с плавающей точкой и т. д. Можно образовать новые типы сообщений из уже существующих.

MPI поддерживает множество операций коммуникации. Ниже приведена операция, которая используется для отправки сообщений:

```
MPI_Send(buffer, count, data_type, destination, tag, communicator)
```

Этот вызов отправляет содержимое буфера (*buffer*) с элементами определенного типа (*data_type*) в пункт назначения. *Count* — это число элементов буфера. Поле *tag* помечает сообщение; получатель может сказать, что он будет принимать

сообщение только с данным тегом. Последнее поле показывает, к какой группе процессов относится целевой процесс (поле *destination* — это просто индекс списка процессов из определенной группы). Соответствующий вызов для получения сообщения таков:

```
MPI_Recv(&buffer, count, data_type, source, tag, communicator, &status)
```

В нем сообщается, что получатель ищет сообщение определенного типа из определенного источника с определенным тегом.

MPI поддерживает 4 основных типа коммуникации. Первый тип синхронный. В нем отправитель не может начать передачу данных, пока получатель не вызовет процедуру `MPI_Recv`. Второй тип — коммуникация с использованием буфера. Ограничение для первого типа здесь недействительно. Третий тип стандартный. Он зависит от реализации и может быть либо синхронным, либо с буфером. Четвертый тип сходен с первым. Здесь отправитель требует, чтобы получатель был доступен, но без проверки. Каждый из этих примитивов бывает двух видов: блокирующим и неблокирующим, что в сумме дает 8 примитивов. Получение может быть только в двух вариантах: блокирующим и неблокирующим.

MPI поддерживает коллективную коммуникацию — широковещание, распределение и сбор данных, обмен данными, агрегацию и барьер. При любых формах коллективной коммуникации все процессы в группе должны делать вызов, причем с совместимыми параметрами. Если этого сделать не удается, возникает ошибка. Например, процессы могут быть организованы в виде дерева, в котором значения передаются от листьев к корню, подчиняясь определенной обработке на каждом шаге (это может быть сложение значений или взятие максимума). Это типичная форма коллективной коммуникации.

Четвертое основное понятие в MPI — **виртуальная топология**, когда процессы могут быть организованы в дерево, кольцо, решетку, тор и т. д. Такая организация процессов обеспечивает способ наименования каналов связи и облегчает коммуникацию.

В MPI-2 были добавлены динамические процессы, доступ к удаленной памяти, неблокирующая коллективная коммуникация, расширяемая поддержка процессов ввода-вывода, обработка в режиме реального времени и многие другие особенности. В научном сообществе идет война между лагерями MPI и PVM. Те, кто поддерживают PVM, утверждают, что эту систему проще изучать и легче использовать. Те, кто на стороне MPI, утверждают, что эта система выполняет больше функций и, кроме того, она стандартизована, что подтверждается официальным документом.

Совместно используемая память на прикладном уровне

Из наших примеров видно, что мультикомпьютеры можно расширить до гораздо больших размеров, чем мультипроцессоры. Sun Enterprise 10000, где максимальное число процессоров — 64, и NUMA-Q, где максимальное число процессоров — 256, являются представителями больших мультипроцессоров UMA и NUMA соответственно. А машины T3E и Option Red содержат 2048 и 9416 процессоров соответственно.

Однако мультикомпьютеры не имеют совместно используемой памяти на архитектурном уровне. Это и привело к появлению таких систем с передачей сообщений, как PVM и MPI. Большинство программистов предпочитают иллюзию совместно используемой памяти, даже если ее на самом деле не существует. Если удастся достичь этой цели, мы сразу получим дешевое аппаратное обеспечение больших размеров плюс обеспечим легкость программирования.

Многие исследователи пришли к выводу, что общая память на архитектурном уровне может быть нерасширяемой, но зато существуют другие способы достижения той же цели. Из рисунка 8.3 видно, что есть и другие уровни, на которых можно ввести совместно используемую память. В следующих разделах мы рассмотрим, как в мультикомпьютере можно ввести совместно используемую память в модель программирования при отсутствии ее на уровне аппаратного обеспечения.

Распределенная совместно используемая память

Один из классов систем с общей памятью на прикладном уровне — это системы со страничной организацией памяти. Такая система называется **DSM (Distributed Shared Memory — распределенная совместно используемая память)**. Идея проста: ряд процессоров в мультикомпьютере разделяет общее виртуальное адресное пространство со страничной организацией. Самый простой вариант — когда каждая страница содержится в ОЗУ ровно для одного процессора. На рис. 8.34, *а* мы видим общее виртуальное адресное пространство, которое состоит из 16 страниц и распространяется на 4 процессора.

Когда процессор обращается к странице в своем локальном ОЗУ, чтение и запись происходят без задержки. Однако если процессор обращается к странице из другого ОЗУ, происходит ошибка из-за отсутствия страницы. Только в этом случае отсутствующая страница берется не с диска. Вместо этого операционная система посылает сообщение в узел, в котором находится данная страница, чтобы преобразовать ее и отправить к процессору. После получения страницы она преобразуется в исходное состояние, а приостановленная команда выполняется заново, как и при обычной ошибке из-за отсутствия страницы. На рис. 8.34, *б* мы видим ситуацию после того, как процессор 0 получил ошибку из-за отсутствия страницы 10, которая была передана из процессора 1 в процессор 0.

Впервые идея была реализована в машине IVY [83, 84]. В результате в мультикомпьютере появилась память совместного использования, согласованная по последовательности. В целях улучшения производительности возможны оптимизации. Первая оптимизация, появившаяся в IVY, — страницы, предназначенные только для чтения, могли присутствовать в нескольких узлах одновременно. В случае ошибки из-за отсутствия страницы в запрашивающую машину посылается копия этой страницы, но оригинал остается на месте, поскольку нет никакой опасности конфликтов. На рисунке 8.34, *в* показана ситуация, когда два процессора делят общую страницу, предназначенную только для чтения (это страница 10).

Но даже при такой оптимизации трудно достичь высокой производительности, особенно когда один процесс записывает несколько слов вверху какой-либо страницы, а другой процесс в другом процессоре в это же время записывает несколько слов внизу той же страницы. Поскольку существует только одна копия этой стра-

ницы, страница постоянно должна передаваться туда и обратно. Эта ситуация называется **ложным совместным использованием**.

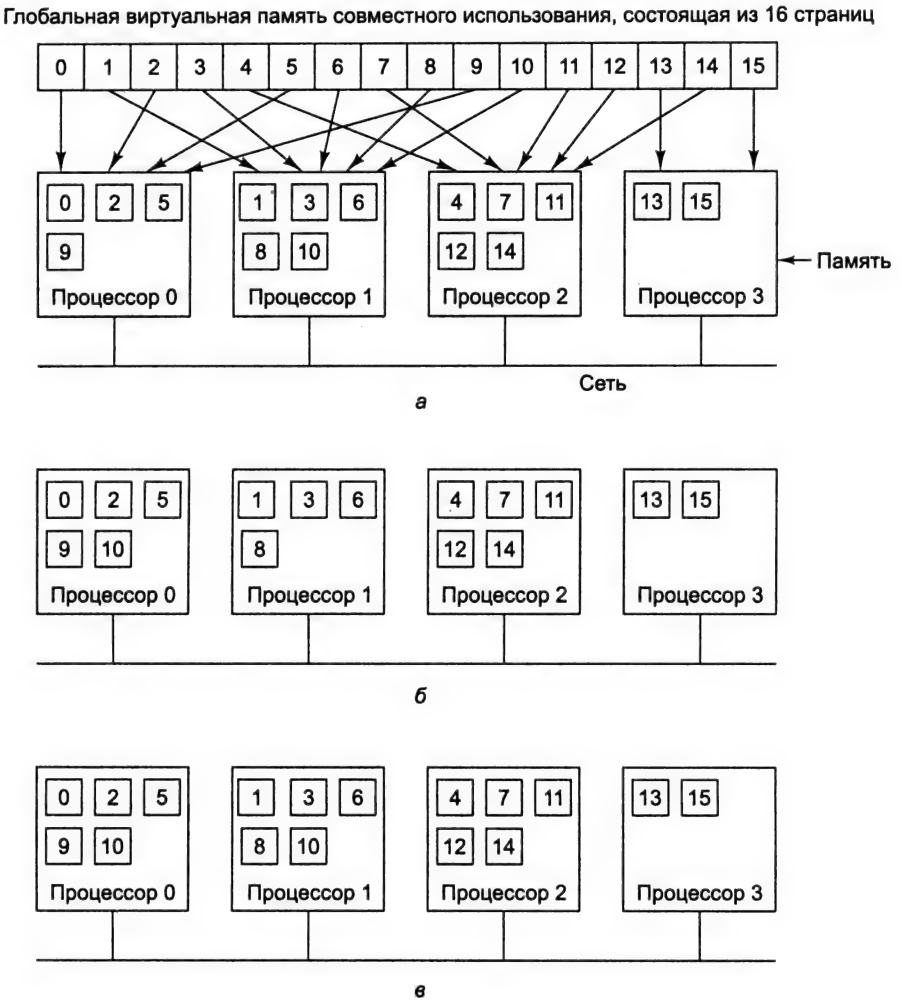


Рис. 8.34. Виртуальное адресное пространство, состоящее из 16 страниц, которые распределены между четырьмя узлами в мультикомпьютере: исходное положение (*а*); положение после обращения процессора 0 к странице 10 (*б*); положение после обращения процессора 0 к странице 10 (*в*); в данном случае эта страница предназначена только для чтения

Проблему ложного совместного использования можно разрешить по-разному. Например, можно отказаться от согласованности по последовательности в пользу свободной согласованности [6]. Потенциально записываемые страницы могут присутствовать в нескольких узлах одновременно, но перед записью процесс должен совершить операцию `acquire`, чтобы сообщить о своем намерении. В этот момент все копии, кроме последней, объявляются недействительными. Нельзя делать

никаких копий до тех пор, пока не будет совершена операция `release` и после этого страница снова станет общей.

Второй способ оптимизации — изначально преобразовывать каждую записываемую страницу в режим «только для чтения». Когда в страницу запись производится впервые, система создает копию страницы, называемую **двойником**. Затем страница преобразуется в формат «для чтения и записи», и последующие записи могут производиться на полной скорости. Если позже произойдет ошибка из-за отсутствия страницы и страницу придется туда доставлять, между текущей страницей и двойником производится пословное сравнение. Пересылаются только те слова, которые были изменены, что сокращает размер сообщений.

Если возникает ошибка из-за отсутствия страницы, нужно определить, где находится отсутствующая страница. Возможны разные способы, в том числе каталоги, которые используются в машинах NUMA и COMA. Многие средства, которые используются в DSM, применимы к машинам NUMA и COMA, поскольку DSM — это программная реализация машины NUMA или COMA, в которой каждая страница трактуется как строка кэш-памяти.

DSM — это обширная область для исследования. Большой интерес представляют системы CASHMERE [76, 141], CRL [68], Shata [124] и Treadmarks [6, 87].

Linda

Системы DSM со страничной организацией памяти (например, IVY и Treadmarks) используют аппаратный блок управления памятью, чтобы закрывать доступ к отсутствующим страницам. Хотя пересылка только отдельных слов вместо всей страницы и влияет на производительность, страницы неудобны для совместного использования, поэтому применяются и другие подходы.

Один из таких подходов — система Linda, в которой процессы на нескольких машинах снабжены структурированной распределенной памятью совместного использования [21, 22]. Доступ к такой памяти осуществляется с помощью небольшого набора примитивных операций, которые можно добавить к существующим языкам (например, C или FORTRAN), в результате чего получатся параллельные языки, в данном случае C-Linda и FORTRAN-Linda.

В основе системы Linda лежит понятие абстрактного **пространства кортежей**, которое глобально по отношению ко всей системе и доступно всем процессам этой системы. Пространство кортежей похоже на глобальную память совместного использования, только с определенной встроенной структурой. Пространство содержит ряд **кортежей**, каждый из которых состоит из одного или нескольких полей. Для C-Linda поля могут содержать целые числа, числа типа long integers, числа с плавающей точкой, а также массивы (в том числе цепочки) и структуры (но не другие кортежи). В листинге 8.4. приведено 3 примера кортежей.

Над кортежами можно совершать 4 операции. Первая из них, `out`, помещает кортеж в пространство кортежей. Например, операция

```
out("abc", 2, 5);
```

помещает кортеж («abc», 2, 5) в пространство кортежей. Поля операции `out` обычно содержат константы, переменные и выражения, например

```
out("matrix-1", i, j, 3.14);
```

Это выражение помещает в пространство кортеж с четырьмя полями, причем второе и третье поля определяются по текущим значениям переменных i и j.

Вторая операция, in, извлекает кортеж из пространства. Обращение к кортежам производится по содержанию, а не по имени или адресу. Поля операции in могут содержать выражения или формальные параметры. Рассмотрим операцию

```
in("abc", 2, ? i);
```

Эта операция отыскивает кортеж, состоящий из цепочки «abc», целого числа 2 и третьего поля, которое может содержать любое целое число (предполагается, что i — это целое число). Найденный кортеж удаляется из пространства кортежей, а переменной i приписывается значение третьего поля. Когда два процесса выполняют одну и ту же операцию in одновременно, успешно выполнит эту операцию только один из них, если нет двух и более подходящих кортежей. Пространство кортежей может содержать много копий одного кортежа.

Листинг 8.4. Три кортежа системы Linda

```
("abc",2,5)
("matrix-1",1,6,3.14)
("family", "is sister", Carolyn, Elinor)
```

Алгоритм соответствия, который используется в операции in, довольно прост. Поля **шаблона** операции in сравниваются с соответствующими полями каждого кортежа в пространстве кортежей. Совпадение происходит, если удовлетворены следующие три условия:

1. Шаблон и кортеж имеют одинаковое количество полей.
2. Типы соответствующих полей совпадают.
3. Каждая константа или переменная в шаблоне совпадает с полем кортежа.

Формальные параметры, указываемые знаком вопроса, за которым следует имя переменной или типа, не участвуют в сравнении (за исключением проверки типа), хотя тем параметрам, которые содержат имя переменной, присваиваются значения после совпадения.

Если соответствующий кортеж отсутствует, процесс приостанавливается, пока другой процесс не вставит нужный кортеж, и в этот момент вызывающий процесс автоматически возобновит работу и найдет этот новый кортеж. Процессы блокируются и разблокируются автоматически, поэтому если один процесс собирается выводить кортеж, а другой — вводить его, то не имеет никакого значения, какой из процессов будет первым.

Третья операция, read, сходна с in, только она не удаляет кортеж из пространства. Четвертая операция, eval, параллельно оценивает свои параметры, а полученный в результате кортеж копируется в пространство кортежей. Этот механизм можно использовать для выполнения любых вычислений. Именно так создаются параллельные процессы в системе Linda.

Основной принцип программирования в системе Linda — это модель **replicated worker**. В основе этой модели лежит понятие **пакета задач** (task bag), которые нужно выполнить. Основной процесс начинается с выполнения цикла, содержащего операцию

```
out("task-bag",job);
```

При каждом прохождении цикла одна из задач выдается в пространство кортежей. Каждый рабочий процесс начинает работу с получения кортежа с описанием задачи, используя операцию

```
in("Task-bag", ?job);
```

Затем он выполняет эту задачу. По выполнении задачи он получает следующую. Новая задача тоже может быть помещена в пакет задач во время выполнения. Таким образом, работа динамически распределяется среди рабочих процессов, и каждый рабочий процесс занят постоянно.

Существуют различные реализации Linda в мультикомпьютерных системах. Во всех реализациях ключевым является вопрос о том, как распределить кортежи по машинам и как их находить. Возможные варианты — широковещание и каталоги. Дублирование — это тоже важный вопрос. Подробнее об этом см. [16].

Orca

Несколько другой подход к совместно используемой памяти на прикладном уровне в мультикомпьютере — в качестве общей совместно используемой единицы использовать полные объекты, а не просто кортежи. Объекты состоят из внутреннего (скрытого) состояния и процедур для оперирования этим состоянием. Поскольку программист не имеет прямого доступа к состоянию, появляется возможность совместного использования ресурсов машинами, которые не имеют общей физической памяти.

Одна из таких систем называется Orca [11, 13, 14]. Orca — это традиционный язык программирования (основанный на Modula 2), к которому добавлены две особенности — объекты и возможность создавать новые процессы. Объект Orca — это стандартный тип данных, аналогичный объекту в языке Java или пакету в языке Ada. Объект заключает в себе внутренние структуры данных и написанные пользователем процедуры, которые называются **операциями**. Объекты пассивны, то есть они не содержат потоков, которым можно посылать сообщения. Процессы получают доступ к внутренним данным объекта путем вызова его процедур.

Каждая процедура в Orca состоит из списка пар (предохранитель (guard), блок операторов). Предохранитель — это логическое выражение, которое может принимать значение «истина» (*true*) или «ложь (*false*). Когда вызывается операция, все ее предохранители оцениваются в произвольном порядке. Если все они ложны (*false*), вызывающий процесс приостанавливается до тех пор, пока один из них не примет значение *true*. При нахождении предохранителя со значением *true* выполняется следующий за ним блок выражений. В листинге 8.5 показан объект *stack* с двумя операциями *push* и *pop*.

Листинг 8.5. Упрощенный объект stack в системе Orca с внутренними данными и двумя операциями

```
Object implementation stack;
    top:integer;                          #хранилище для стека
    stack:array[integer 0..N-1]of integer;

    operation push(item:integer);         #функция, которая ничего не
    begin                                 #возвращает
        stack[top]:=item;                 #помещаем элемент в стек
        top:=top+1;                       #увеличения указателя стека
    end;
```

```
    operation pop():integer;                    #функция, которая возвращает
    begin                                       #целое число
        guard top>0 do                          #приостанавливает работу, если стек пуст
            top:=top-1;                         #уменьшает указатель стека
            return stack[top];                  # возвращает вершину стека
        od;
    end;

begin
    top:=0;                                     #инициализация
end;
```

Как только определен объект *stack*, нужно объявить переменные этого типа:

```
s, t, stack;
```

Такая запись создает два стековых объекта и устанавливает переменную *top* в каждом объекте на 0. Целочисленную переменную k можно поместить в стек s с помощью выражения

```
s$push(k);
```

и т. д. Операция *pop* содержит предохранитель, поэтому попытка вытолкнуть переменную из пустого стека вызовет блокировку вызывающего процесса до тех пор, пока другой процесс не положит что-либо в стек.

Orca содержит **оператор ветвления** (fork statement) для создания нового процесса в процессоре, определяемом пользователем. Новый процесс запускает процедуру, названную в команде ветвления. Параметры, в том числе объект, могут передаваться новому процессу. Именно так объекты распределяются среди машин. Например, выражение

```
for i in 1..n do fork foobar(s) on i; od;
```

порождает новый процесс на каждой из машин с 1 по n, запуская программу *foobar* в каждой из них. Поскольку эти n новых процессов (а также исходный процесс) работают параллельно, они все могут помещать элементы в общий стек s и выталкивать элементы из общего стека s, как будто все они работают на мультипроцессоре с памятью совместного использования.

Операции в объектах совместного использования атомарны и согласованы по последовательности. Если несколько процессов выполняют операции над одним общим объектом практически одновременно, система выбирает определенный порядок выполнения, и все процессы «видят» этот же порядок.

В системе Orca данные совместного использования совмещаются с синхронизацией не так, как в системах со страничной организацией памяти. В программах с параллельной обработкой нужны два вида синхронизации. Первый вид — взаимное исключение. Этот метод не позволяет двум процессам одновременно выполнять одну и ту же критическую область. В системе Orca каждая операция над общим объектом похожа на критическую область, поскольку система гарантирует, что конечный результат будет таким же, как если бы все критические области выполнялись последовательно одна за другой. В этом отношении объект Orca похож на распределенное контролирующее устройство [61].

Второй вид синхронизации — условная синхронизация, при которой процесс блокируется и ждет выполнения определенного условия. В системе Orca условная синхронизация осуществляется при помощи предохранителей. В примере, приве-

денном в листинге 8.5, процесс, который пытается вытолкнуть элемент из пустого стека, блокируется до появления в стеке элементов.

В системе Orca допускается копирование объектов, миграция и вызов операций. Каждый объект может находиться в одном из двух состояний: он может быть единственным, а может быть продублирован. В первом случае объект существует только на одной машине, поэтому все запросы отправляются туда. Продублированный объект присутствует на всех машинах, которые содержат процесс, использующий этот объект. Это упрощает операцию чтения (поскольку ее можно производить локально), но усложняет процесс обновления. При выполнении операции, которая изменяет продублированный объект, сначала нужно получить от центрального процесса порядковый номер. Затем в каждую машину, содержащую копию объекта, отправляется сообщение о необходимости выполнить эту операцию. Поскольку все такие обновления обладают порядковыми номерами, все машины просто выполняют операции в порядке номеров, что гарантирует согласованность по последовательности.

Globe

Большинство систем DSM, Linda и Orca работают в пределах одного здания или предприятия. Однако можно построить систему с совместно используемой памятью на прикладном уровне, которая может распространяться на весь мир. В системе Globe объект может находиться в адресном пространстве нескольких процессов одновременно, возможно, даже на разных континентах [72, 154]. Чтобы получить доступ к данным общего объекта, пользовательские процессы должны пройти через его процедуры, поэтому для разных объектов возможны разные способы реализации. Например, можно иметь один экземпляр данных, который запрашивается по мере необходимости (это удобно для данных, часто обновляемых одним владельцем). Другой вариант — когда все данные находятся в каждой копии объекта, а сигналы об обновлении посылаются каждой копии в соответствии с надежным протоколом широковещания.

Цель системы Globe — работать на миллиард пользователей и содержать триллион объектов — делает эту систему амбициозной. Ключевыми моментами являются размещение объектов, управление ими, а также расширение системы. Система Globe содержит общую сеть, в которой каждый объект может иметь собственную стратегию дублирования, стратегию защиты и т. д.

Среди других широкомасштабных систем можно назвать Globus [40, 41] и Legion [50, 51], но они, в отличие от Globe, не создают иллюзию совместного использования памяти.

Краткое содержание главы

Компьютеры параллельной обработки можно разделить на две основные категории: SIMD и MIMD. Машины SIMD выполняют одну команду одновременно над несколькими наборами данных. Это массивно-параллельные процессоры и векторные компьютеры. Машины MIMD выполняют разные программы на разных машинах. Машины MIMD можно подразделить на мультипроцессоры, которые

совместно используют общую основную память, и мультикомпьютеры, которые не используют общую основную память. Системы обоих типов состоят из процессоров и модулей памяти, связанных друг с другом различными высокоскоростными сетями, по которым между процессорами и модулями памяти передаются пакеты запросов и ответов. Применяются различные топологии, в том числе решетки, торы, кольца и гиперкубы.

Для всех мультипроцессоров ключевым вопросом является модель согласованности памяти. Из наиболее распространенных моделей можно назвать согласованность по последовательности, процессорную согласованность, слабую согласованность и свободную согласованность. Мультипроцессоры можно строить с использованием отслеживающей шины, например, в соответствии с протоколом MESI. Кроме того, возможны различные сети, а также машины на основе каталога NUMA и COMA.

Мультипроцессоры можно разделить на системы MPP и COW, хотя граница между ними произвольна. К системам MPP относятся Cray T3E и Intel/Sandia Option Red. В них используются запатентованные высокоскоростные сети межсоединений. Системы COW, напротив, строятся из таких стандартных деталей, как Ethernet, ATM и Myrinet.

Мультикомпьютеры часто программируются с использованием пакета с передачей сообщений (например, PVM или MPI). Оба пакета поддерживают библиотечные вызовы для отправки и получения сообщений. Оба работают поверх существующих операционных систем.

Альтернативный подход — использование памяти совместного использования на прикладном уровне (например, система DSM со страничной организацией, пространство кортежей в системе Linda, объекты в системах Orca и Globe). Система DSM моделирует совместно используемую память на уровне страниц, и в этом она сходна с машиной NUMA. Системы Linda, Orca и Globe создают иллюзию совместно используемой памяти с помощью кортежей, локальных объектов и глобальных объектов соответственно.

Вопросы и задания

1. Утром пчелиная матка созывает рабочих пчел и сообщает им, что сегодня им нужно собрать нектар ноготков. Рабочие пчелы вылетают из улья и летят в разных направлениях в поисках ноготков. Что это за система — SIMD или MIMD?
2. Вычислите диаметр сети для каждой топологии, изображенной на рис. 8.4.
3. Для каждой топологии, изображенной на рис. 8.4, определите степень отказоустойчивости. Отказоустойчивость — максимальное число каналов связи, после утраты которых сеть не будет разделена на две части.
4. Рассмотрим топологию двойной тор (см. рис. 8.4, е), расширенную до размера k×k. Каков диаметр такой сети? Подсказка: четное и нечетное k нужно рассматривать отдельно.

5. Представим сеть в форме куба 8×8×8. Каждый канал связи имеет дуплексную пропускную способность 1 Гбайт/с. Какова бисекционная пропускная способность этой сети?
6. Рассмотрим сеть в форме прямоугольной решетки размером 4 коммутатора в ширину и 3 коммутатора в высоту. В ней пакеты, исходящие из левого верхнего угла и направляющиеся в правый нижний угол, могут следовать по любому из нескольких возможных путей. Пронумеруйте верхний ряд коммутаторов с 1 по 4, следующий ряд — с 5 по 8, а нижний — с 9 по 12. Выпишете все пути, по которым пакеты перемещаются только вправо или вниз, начиная с коммутатора 1 и заканчивая коммутатором 12.
7. Закон Амдала ограничивает потенциальный коэффициент ускорения, достижимый в компьютере параллельного действия. Вычислите как функцию от f максимально возможный коэффициент ускорения, если число процессоров стремится к бесконечности. Каково значение этого предела для f=0,1?
8. На рисунке 8.10 показано, что расширение невозможно с шиной, но возможно с решеткой. Каждая шина или канал связи имеет пропускную способность b. Вычислите среднюю пропускную способность на каждый процессор для каждого из четырех случаев. Затем расширьте каждую систему до 64 процессоров и выполните те же вычисления. Чему равен предел, если число процессоров стремится к бесконечности?
9. Компьютерная компания выпускает системы, состоящие из n компьютеров с совместно используемой памятью, организованных в квадратную решетку. Однажды вице-президенту компании приходит в голову идея выпустить новый продукт: трехмерную решетку, в которой n компьютеров организованы в правильный куб (это возможно, например, для n=4096).
    1. Как это изменение повлияет на максимальное время ожидания?
    2. Как это изменение повлияет на общую пропускную способность?
10. Когда мы говорили о согласованности памяти, мы сказали, что модель согласованности — это вид контракта между программным обеспечением и памятью. Зачем нужен такой контракт?
11. Векторный процессор (например, Cray-1) содержит арифметические устройства с конвейерами из четырех стадий. Прохождение каждой стадии занимает 1 нс. Сколько времени понадобится для сложения двух векторов из 1024 элементов?
12. Рассмотрим мультипроцессор с общей шиной. Что произойдет, если два процессора попытаются получить доступ к глобальной памяти в один и тот же момент?
13. Предположим, что по техническим причинам отслеживающий кэш может отслеживать только адресные линии и не может отслеживать информационные. Повлияет ли это изменение на протокол сквозной записи?
14. Рассмотрим простую модель мультипроцессорной системы с шиной без кэш-памяти. Предположим, что одна из каждых четырех команд обращается к памяти и что при каждом обращении к памяти шина занимается на все

время выполнения команды. Если шина занята, то запрашивающий процессор становится в очередь «первым вошел — первым вышел». Насколько быстрее будет работать система с 64 процессорами по сравнению с однопроцессорной системой?

15. Протокол MESI содержит 4 состояния. Каким из 4 состояний можно пожертвовать и каковы будут последствия каждого из четырех вариантов? Если бы вам пришлось выбрать только три состояния, какие бы вы выбрали?
16. Бывают ли в протоколе MESI такие ситуации, когда строка кэш-памяти присутствует в локальной кэш-памяти, но при этом все равно требуется транзакция шины? Если да, то опишите такую ситуацию.
17. Предположим, что к общей шине подсоединено n процессоров. Вероятность, что любой процессор пытается использовать шину в данном цикле, равна p. Какова вероятность, что:
    1. Шина свободна (0 запросов).
    2. Совершается ровно один запрос.
    3. Совершается более одного запроса.
18. Процессоры Sun Enterprise 10000 работают с частотой 333 МГц, а отслеживающие шины — с частотой 83,3 МГц. Если имеется 64 процессора, ясно, что шины не справятся с такой нагрузкой. Тем не менее машина работает. Объясните, почему.
19. В этой книге мы вычислили, что производительности координатного коммутатора было достаточно для обработки 167 млн отслеживаний/с, когда к нему подсоединено 16 плат, причем мы даже принимаем во внимание тот факт, что из-за конфликтных ситуаций на практике пропускная способность составляет 60% от теоретической. Небольшая машина Enterprise 10000 может содержать всего 4 платы (16 процессоров). Будет ли такая машина работать с полной скоростью?
20. Предположим, что провод между коммутатором 2A и коммутатором 3B в сети Omega поврежден. Какие именно элементы будут отрезаны друг от друга?
21. Области памяти, к которым часто происходят обращения, представляют большую проблему в многоступенчатых сетях. А являются ли они проблемой в системах с шинной организацией?
22. Сеть Omega соединяет 4096 процессоров RISC, время цикла каждого из которых составляет 60 нс, с 4096 бесконечно быстрыми модулями памяти. Каждый переключательный элемент дает задержку 5 нс. Сколько пустых циклов требуется для команды `LOAD`?
23. Рассмотрим машину с сетью Omega (см. рис. 8.22). Предположим, что программа и стек i хранятся в модуле памяти i. Какое незначительное изменение топологии может сильно повлиять на производительность? (IBM RP3 и BBN Butterfly используют эту измененную топологию.) Какой недостаток имеет новая топология по сравнению со старой?

24. В мультипроцессоре NUMA обращение к локальной памяти занимает 20 нс, а к памяти другого процессора — 120 нс. Программа во время выполнения совершает N обращений к памяти, 1% из которых — обращения к странице P. Изначально эта страница находится в удаленной памяти, а на копирование ее из локальной памяти требуется C нс. При каких обстоятельствах эту страницу следует копировать локально, если ее не используют другие процессоры?
25. Система DASH содержит b байтов памяти, которые распределены между n кластерами. В каждом кластере содержится p процессоров. Размер строки кэш-памяти составляет c байтов. Напишите формулу для общего количества памяти, предоставленного каталогам (исключая два бита состояния на каждый элемент каталога).
26. Рассмотрим мультипроцессор CC-NUMA (см. рис. 8.24), содержащий 512 узлов по 8 Мбайт каждый. Если длина строки кэш-памяти составляет 64 байта, каков процент непроизводительных затрат для каталогов? Как повлияет увеличение числа узлов на непроизводительные затраты (они увеличатся, уменьшатся или останутся без изменений)?
27. На какую операцию в SCI требуется больше всего времени?
28. Мультипроцессор на базе SCI содержит 63 узла. Длина строки кэш-памяти составляет 32 байта, а общее адресное пространство составляет 2^{32} байтов. Размер кэш-памяти в каждом узле — 1 Мбайт. Сколько байтов нужно иметь в каждом кэш-каталоге?
29. Машина NUMA-Q 2000 содержит 63 узла. Предложите ввести 64 узла вместо 63. Почему компания Sequent в качестве максимума выбрала именно 63, а не 64?
30. В этой книге мы обсуждали 3 варианта примитива send — синхронный, блокирующий и неблокирующий. Предложите четвертый метод, который схож с блокирующей операцией send, но немного отличается по свойствам. Какое преимущество и какой недостаток имеет новый метод по сравнению с обычной блокирующей операцией send?
31. Рассмотрим компьютер, который работает в сети с аппаратным широковещанием (например, Ethernet). Почему важно соотношение операций чтения (которые не изменяют внутреннее состояние переменных) и операций записи (которые изменяют внутреннее состояние переменных)?
32. Многие вопросы, возникающие при разработке мультипроцессоров, возникают также при разработке совместно используемой памяти на прикладном уровне. Один из таких вопросов — выбор одной из двух политик: обновление или объявление недействительным. Какая политика используется в системе Orca?

Глава 9
Библиография

В предыдущих главах мы с разной степенью подробности обсуждали достаточно широкий ряд вопросов. Эта глава предназначена для читателей, которые заинтересованы в более глубоком изучении строения компьютеров. В разделе «Литература для дальнейшего чтения» содержится список литературы к каждой главе. В разделе «Алфавитный список литературы» приводится алфавитный список всех книг и статей, на которые мы ссылались в этой книге.

Литература для дальнейшего чтения

Вводная и неспециальная литература

1. Hamacher et al., *Computer Organization*, 4th ed.

   Традиционный учебник об организации компьютеров (процессоры, память, ввод-вывод, арифметика, периферийные устройства). Основные примеры — 68000 и Power PC.

2. Hayes, *Computer Architecture and Organization*, 3rd ed.

   Еще одна традиционная книга о компьютерной организации с уклоном к аппаратному обеспечению. В книге рассматриваются процессоры, тракт данных, микропрограммирование, конвейеры, организация памяти и процесс ввода-вывода.

3. Patterson and Hennessy, *Computer Organization and Design*.

   Эта книга почти на 1000 страниц содержит обширный материал о компьютерной архитектуре, в частности о разработке процессоров RISC. Упор делается на то, как достичь высокой производительности с помощью конвейеризации и других технологий.

4. Price, *A History of Calculating Machines*.

   Современные компьютеры восходят к машине Бэббиджа, созданной в XIX веке, но люди производили различные вычисления с самого начала цивилизации. В этой иллюстрированной статье прослеживается вся история счета, математики, календарей и вычислений с 3000 г. до н. э. до начала XX века.

5. Slater, *Portraits in Silicon.*

   Почему Деннис Ритчи защитил докторскую диссертацию в Гарварде? Почему Стив Джобс стал вегетарианцем? Ответы на эти и другие вопросы вы можете найти в этой увлекательной книге. Книга содержит 34 короткие биографии людей, которые сформировали компьютерную промышленность (от Чарльза Бэббиджа до Дональда Кнута).

6. Stallings, *Computer Organization and Archirecture*, 4th ed.

   Книга по компьютерной архитектуре. В ней рассматриваются и те вопросы, которые мы обсуждали в нашей книге.

7. Wilkes, *Computers Then and Now.*

   Автор книги Морис Уилкс, один из первых компьютерных разработчиков и изобретатель микропрограммирования, излагает историю компьютеров с 1946 по 1968 год. Он рассказывает о войне между приверженцами автоматического программирования («space cadets») и традиционалистами, которые предпочитали программировать в восьмеричной системе.

Организация компьютерных систем

1. Ng, *Advances in Disk Technology: Performance Issues.*

   В течение последних 20 лет специалисты предсказывают устаревание магнитных дисков. Однако они все еще в ходу. В этой работе говорится о том, что технологии магнитных дисков стремительно развиваются и что они будут служить нам еще долгие годы.

2. Messmer, *The Indispensable PC Hardware Book*, 3rd ed.

   Толстая книга на 1384 страницы (36 глав и 13 приложений). Здесь очень подробно описываются процессоры 80x86, память, шины, вспомогательные микросхемы и периферийные устройства. Если вы уже прочитали книгу Нортона и Гудмэна (см. ниже) и хотите получить более подробную информацию, обратитесь к этой работе.

3. Norton and Goodman, *Inside the PC*, 7th ed.

   Большинство книг по аппаратному обеспечению написаны для инженеров-электриков, и их сложно читать тем, кто занимается программным обеспечением. Эта книга не такая. В ней просто и доступно рассказывается об аппаратном обеспечении персональных компьютеров. Речь идет о процессоре, памяти, шинах, дисках, мониторах, устройствах ввода-вывода, переносных персональных компьютерах, сетях и т. п. Очень редкая и ценная книга.

4. Pilgrim, *Build Your Own Pentium II PC.*

   Если вы умеете обращаться с отверткой и паяльником и хотите сконструировать свой собственный компьютер из отдельных деталей, эта книга может вам пригодиться. Даже если вы намерены купить компьютер в магазине, в этой книге вы можете найти полезную информацию о компонентах персонального компьютера, о том, как они работают и какие у них особенности и дополнительные возможности.

Цифровой логический уровень

1. Floyd, *Digital Fundamentals*, 6th ed.

   Эта огромная иллюстрированная книга вполне подойдет для тех, кто хочет более подробно изучить цифровой логический уровень. В ней рассказывается о комбинационных логических схемах, программируемых логических устройствах, триггерах, сдвиговых регистрах, о памяти, интерфейсах и многом другом.

2. Katayama, *Trends in Semiconductor Memory*.

   Хотя память работает гораздо медленнее процессоров, полупроводниковая память все же развивается. В этой статье рассказывается о некоторых тенденциях развития динамического ОЗУ.

3. Mano and Kime, *Logic and Computer Design Fundamentals*.

   Эта книга не обладает такой проработанностью и ясностью, как книга Флойда (Floyd), но тоже является неплохим пособием по цифровому логическому уровню. В ней содержится информация о комбинационных и последовательных схемах, регистрах, памяти, процессорах и устройствах ввода-вывода.

4. Mazidi and Mazidi, *The 80x86 IBM PC and Compatible Computers*, 2nd ed.

   Книга предназначена для читателей, которые интересуются устройством всех микросхем персонального компьютера. В книге содержатся целые главы об основных микросхемах, а также масса прочей информации об аппаратном обеспечении и программировании на языке ассемблера.

5. McKee et al., *Smarter Memory: Improving Bandwidth for Streamed References*.

   По сравнению с процессорами память с течением десятилетий работает все медленнее и медленнее. В этой работе рассматриваются различные вопросы, связанные с производительностью памяти, а также возможности решения этой проблемы.

6. Nelson et al., *Digital Logic and Circuit Analysis and Design*.

   Еще одна всеобъемлющая книга по цифровой логике. В ней подробно рассказывается о последовательных и комбинационных схемах.

7. Triebel, *The 80386, 80486 and Pentium Processor*.

   Эта книга имеет отношение и к аппаратному, и к программному обеспечению, а также к интерфейсам. В ней рассказывается все о процессорах, памяти, устройствах ввода-вывода и о сопряжении микросхем компьютера 80x86, а также о том, как их программировать на языке ассемблера. В ней всего 915 страниц, но она содержит столько же материала, как и книга Мессмера, поскольку страницы здесь больше по размеру.

Микроархитектурный уровень

1. Handy, *The Cache Memory Book*, 2nd ed.

   Вопрос разработки кэш-памяти очень важен, поэтому существуют целые книги, посвященные этому вопросу. В этой книге обсуждаются логическая и физическая кэш-память, размер строки, сквозная и обратная запись,

объединенная и разделенная кэш-память, а также некоторые вопросы программного обеспечения. Целая глава посвящена когерентности кэш-памяти в мультипроцессоре.

2. Johnson, *Superscalar Microprocessor Design.*

   Если вы интересуетесь подробностями разработки суперскалярных процессоров, вам нужно начать именно с этой книги. В ней рассказывается о вызове и декодировании команд, о выдаче команд с изменением последовательности, переименовании регистров, резервациях, прогнозировании переходов и о многом другом.

3. Normoyle et al. *UltraSPARC IIi: Expending the Boundaries of a System on a Chip.*

   UltraSPARC Iii — это версия UltraSPARC II с шиной PCI. В этом труде разработчики рассказывают о том, как работает эта система.

4. McChan and O'Connor, *picoJava: A Direct Execution Engine for Java ByteCode.*

   Эта статья представляет собой краткое введение в микроархитектуру picoJava (и следовательно, микросхемы microJava 701). В ней дана блок-схема, обсуждаются вопросы конвейеризации и рассказывается о различных способах оптимизации.

5. Shriver and Smith, *The Anatomy of a High-Performance Microprocessor.*

   Эта книга хорошо подходит для детального изучения современного процессора на микроархитектурном уровне. Подробно описывается микросхема AMD K6 (клон Pentium). Рассказывается о конвейерах, планировании выполнения команд и о способах повышения производительности.

6. Sima, *Superscalar Instruction Issue.*

   Суперскалярная подача команд чрезвычайно важна в современных процессорах. В этой книге мы затронули некоторые вопросы, связанные с этим (в частности, переименование и спекулятивное выполнение). В статье рассматриваются эти и многие другие вопросы.

7. Wilson, *Challenges and Trends in Processor Design.*

   Неужели разработка процессоров не продвигается? Шесть ведущих разработчиков процессоров из компаний Sun, Cyrix, Motorola, Mips, Intel и Digital рассказывают о перспективах развития процессоров в следующие несколько лет. В 2008 году читать это будет смешно, но в настоящее время ее стоит прочитать.

Уровень команд

1. Antonakos, *The Pentium Microprocessor.*

   Первые девять глав этой книги посвящены тому, как программировать Pentium на языке ассемблера. В последних двух рассказывается об аппаратном обеспечении машины Pentium. Приводятся многочисленные фрагменты программ. Рассматривается базовая система ввода-вывода.

2. Paul, *SPARC Architecture, Assembly Language, Programming, and C.*

   Удивительно, но эта книга по программированию на языке ассемблера посвящена вовсе не серии Intel 80x86. Здесь рассказывается о компьютере SPARC и о том, как программировать на нем.

3. Weaver and Germond, *The SPARC Architecture Manual.*

   В связи с интернационализацией компьютерной промышленности стандарты приобретают особую важность. В этой книге дается определение Version 9 SPARC, а также подробно рассказывается о том, что представляет собой стандарт. В книге содержится много информации о том, как работают 64-битные процессоры SPARC.

Уровень операционной системы

1. Hart, *Win32 System Programming.*

   В отличие от большинства книг по Windows, эта посвящена вовсе не графическому пользовательскому интерфейсу. В ней основное внимание уделяется системным вызовам Windows и тому, как они используются для доступа к файлам, управления памятью и процессами, осуществления взаимодействия между процессами, управления потоками, процессами ввода-вывода и т. д.

2. Jacob and Mudge, *Virtual Memory: Issues of Implementation.*

   Хорошая книга о современной виртуальной памяти. В ней рассказывается о таблицах страниц и TLB на примере MIPS, Power PC и процессоров Pentium.

3. Korn, *Porting UNIX to Windows NT.*

   На первый взгляд может показаться, что переносить программы UNIX на NT легко, поскольку система NT содержит очень много системных вызовов. Однако практика показывает, что сделать это не так-то просто. Автор статьи рассказывает, почему возникают трудности.

4. McKusick et al., *Design and Implementation of the 4.4 BSD Operating System.*

   В отличие от большинства книг по UNIX, эта начинается с фотографии четырех авторов на конференции USENIX Conference. Трое из них много написали о пакете BSD версии 4.4 и высококвалифицированы в этом вопросе. В книге говорится о системных вызовах, процессах, о процессе ввода-вывода. Целый раздел посвящен сетям.

5. Ritchie and Thompson, *The UNIX Time-Sharing System.*

   Это самая первая работа, посвященная системе UNIX. И тем не менее ее стоит прочитать. Из этого маленького зернышка выросла большая операционная система.

6. Solomon, *Inside Windows NT*, 2nd ed.

   Если вы хотите знать, как работает система NT, эта книга для вас. В ней обсуждаются архитектура и механизмы системы, процессы, потоки, управление памятью, защита, процесс ввода-вывода, кэш-память, файловые системы и многие другие вопросы.

7. Tanenbaum and Woodhull, *Operating Systems: Design and Implementation*, 2nd ed.

   Большинство книг об операционных системах касаются только теоретических вопросов. В этой книге теория проиллюстрирована на примере реального программного кода операционной системы MINIX, сходной с UNIX, которая работает на IBM PC и других компьютерах. Исходный код с подробными комментариями приводится в приложении.

Уровень языка ассемблера

1. Irvine, *Assembly Language for Intel-Based Computers*, 3rd ed.

   Тема этой книги — программирование процессоров Intel на языке ассемблера. В ней также рассказывается о программировании ввода-вывода, макросах, файлах, связывании, прерываниях и т. д.

2. Saloman, *Assemblers and Loaders*.

   Все, что вы хотели знать об однопроходных и двупроходных ассемблерах, а также о том, как работают компоновщики и загрузчики, макросы и условная компоновка программы.

Архитектуры компьютеров параллельного действия

1. Adve and Gharachorloo, *Shared Memory Consistency Models: A Tutorial*.

   Во многих современных компьютерах, особенно мультипроцессорах, поддерживается более слабая модель памяти, чем согласованность по последовательности. В этом учебном пособии обсуждаются различные модели и объясняется, как они работают. Здесь также приводятся и опровергаются многочисленные мифы о слабо согласованной памяти.

2. Almasi and Gottlieb, *Highly Parallel Computing*, 2nd ed.

   В этой книге рассказывается о параллельной вычислительной обработке, в том числе о сетях, архитектуре, компиляторах, моделях и приложениях. В ней представлены проблемы аппаратного и программного обеспечения, а также прикладные вопросы.

3. Hill, Multiprocessors Should Support Simple Memory-Consistency Models.

   Слабые модели памяти — важная и спорная проблема в разработке памяти мультипроцессора. Слабые модели допускают определенные оптимизации аппаратного обеспечения (например, совершение обращений к памяти в другом порядке), но усложняют программирование. В этой статье автор обсуждает различные вопросы, связанные с согласованностью памяти, и приходит к выводу, что слабо согласованная память создает больше проблем, чем преимуществ.

4. Hwang and Xu, *Scalable Parallel Computing*.

   Авторы рассматривают и программное, и аппаратное обеспечение, поэтому им удалось дать всестороннее, но доступное описание параллельной вы-

числительной обработки. В книге говорится о мультипроцессорах UMA и NUMA, системах MPP и COW, о передаче сообщений и параллельном программировании.

5. Pfister, *In Search of Clusters*, 2nd ed.

   Хотя определение кластера появляется только на 72-й странице (группа компьютеров, работающих вместе), он, очевидно, включает в себя все обычные мультикомпьютерные и мультипроцессорные системы. Подробно рассматриваются их аппаратное и программное обеспечение, производительность и доступность. Предупредим читателя: хотя стиль автора кажется поначалу увлекательным, к 500-й странице вся увлекательность исчезает.

6. Snir et al., *MPI: The Complete reference Manual.*

   Название книги говорит само за себя. Если вы хотите научиться программировать на MPI, обратитесь к ней. В книге рассказывается о двухточечной и коллективной коммуникации, коммуникаторах, об управлении средой и о многом другом.

7. Stenstrom et al., Trends in Shared Memory Multiprocessing.

   Мультипроцессоры с памятью совместного использования часто считают суперкомпьютерами для сложных научных вычислений. В действительности это только крошечная часть их рынка. В статье обсуждается, какие сферы охватывает рынок таких машин и каково значение их архитектуры.

Двоичные числа и числа с плавающей точкой

1. Cody, *Analysis of Proposals for the Floating-Point Standard.*

   Несколько лет назад Институт инженеров по электротехнике и электронике (IEEE) разработал архитектуру с плавающей точкой, которая стала стандартом *de facto* для всех современных процессоров. Автор обсуждает различные вопросы, предложения и возражения, которые возникали во время процесса стандартизации.

2. Garner, *Nubmer Systems and Arithmetic.*

   Учебное пособие о понятиях двоичной арифметики (в том числе о распространении переноса, системах избыточных чисел, системах остаточных классов и о нестандартном умножении и делении). Особенно рекомендуется для тех, что считает, что узнал все об арифметике в шестом классе.

3. IEEE, *Proc. of the n-th Symposium on Computer Arithmetic.*

   Вопреки общепринятому мнению арифметика является активной областью исследования. Специалистами написано много научных трудов. На симпозиуме обсуждаются проблемы прогрессий, развитие высокоскоростного сложения и умножения, арифметическое аппаратное обеспечение СБИС, сопроцессоры, отказоустойчивость, округление и многие другие вопросы.

4. Knuth, *Seminumerical Algorithms*, 3rd ed.

   Обширный материал о позиционных системах счисления, арифметике с плавающей точкой, арифметике с многократно увеличенной точностью и о случайных числах. Книга требует и заслуживает внимательного изучения.

5. Wilson, *Floating-Point Survival Kit.*

   Хорошая книга для начинающих о числах с плавающей точкой и о стандартах. Обсуждаются некоторые популярные задачи с плавающей точкой (например, *Linpack*).

Алфавитный список литературы

1. *Adams, G. B. III, Agrawal, D. P., and Siegel, H. J.* «A Survey and Comparison of Fault-Tolerant Multistage Interconnection Networks», IEEE Computer Magazine, vol. 20, p. 14–27, June 1987.

2. *Adve, S. V., and Charachorloo, K.* «Shared Memory Consistency Models: A Tutorial», IEEE Computer Magazine, vol. 29, p. 66–76, Dec. 1996.

3. *Adve, S. V., and Hill, M.* «Weak Ordering: A New Definition», Proc. 17th Ann. Int'l. Symp. on Computer Arch., ACM, p. 2–14, 1990.

4. *Agerwala, T., and Cocke, J.* «High Performance Reduced Instruction Set Processors», IBM T.J. Watson Research Center Technical Report RC12434, 1987.

5. *Almasi, G. S., and Gottlieb, A.* Highly Parallel Computing, 2nd ed. Redwood City, CA: Benjamin/Cummings, 1994.

6. *Amza, C., COX, A., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W.* «TreadMarks: Shared Memory Computing on a Network of Workstations», IEEE Computer Magazine, vol. 29, p. 18–28, Feb. 1996.

7. *Anderson, D.* Universal Serial Bus System Architecture, Reading, MA: Addison-Wesley, 1997.

8. *Anderson, T. E., Culler, D. E., Patterson, D. A., and the NOW team* «A Case for NOW (Networks of Workstations)», IEEE Micro Magazine, vol. 15, p. 54–64, Feb. 1995.

9. *Antonakos, J. L.* The Pentium Microprocessor, Upper Saddle River, NJ: Prentice Hall, 1997.

10. *August, D. I., Connors, D. A., Mshlke, S. A., SIAS, J. W., Crozier, K. M., Cheng, B.-C., Eaton, P. R., Olaniran, Q. B., and HWU, W.-M.* «Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture», Proc. 25th Ann. Int'l. Symp. on Computer Arch., ACM, p. 227–237, 1998.

11. *Bal, H. E.* Programming Distributed Sysytems, Hemel Hempstead, England: Prentice Hall Int'l., 1991.

12. *Bal, H. E., Bhoedjang, R., Hofman, R, Jacobs, C., Langendoen, K., Ruhl, T., and Kaashoek, M. F.* «Performance Evaluation of the Orca Shared Object System», ACM Trans. on Computer Systems, vol. 16, p. 1–40, Feb. 1998.

13. *Bal, H. E., Kaashoek, M.F., and Tanenbaum, A. S.* «Orca: A Language for Parallel Programming of Distributed Systems», IEEE trans. on Software Engeneering, vol. 18, p. 190–205, March 1992.

14. *Bal, H. E., and Tanenbaum, A. S.* «Distributed Programming with Shared Data», Proc. 1988 Int'l. Conf. on Computer Languages, IEEE, p. 82–91, 1988.

15. *Bhuyan, L. N., Yang, Q., and Agrawal, D. P.* «Performance of Multiprocessor Interconnection Networks», IEEE Computer Magazine, vol. 22, p. 25–37, Feb. 1989.

16. *Bjornson, R. D.* «Linda on Distributed Memory Multiprocessors», Ph. D. Thesis, Yale Univ., 1993.

17. *Blumrich, M. A., Dubnicki, C., Felten, E. W., Li, K., and Mesarina, M. R.* «Virtual-Memory Mapped Network Interfaces», IEEE Micro Magazine, vol. 15, p. 21–28, Feb. 1995.

18. *Boden, N.J., Cohen, D., Felderman, R. E., Kulawik, A. E., Seitz, C. L., Seizovic, J. N., and Su, W. -K.* «Myrinet: A Gigabit per second Local Area Network», IEEE Micro Magazine, vol. 15, p. 29–36, Feb. 1995.

19. *Bouknight, W.J., Denenberg, S. A., Mcintyre, D. E., Randall, J. M., Sameh, A. H., and Slotnick, D. L.* «The Illiac IV System», Proc. IEEE, p. 369–388, April 1972.

20. *Burkhardt, H., Frank, S., Knobe, B., and Rothnie, J.* «Overview of the KSR-1 Computer Sysytem», Technical Report KSR-TR-9202001, Kendall Square Research Corp, Cambridge, MA, 1992.

21. *Carriero, N., and Gelernter, D.* «The S/Net's Linda Kernel», ACM Trans. on Coputer Sysytems, vol. 4, p. 110–129, May 1986.

22. *Carriero, N., and Gelernter, D.* «Linda and Context», Commun. of the ACM, vol. 32, p. 444–458, April 1989.

23. *Charlesworth, A.* «Starfire: Extending the SMP Envelope», IEEE Micro Magazine, vol. 18, 39–49, Jan./Feb. 1998.

24. *Charlesworth, A., Phelps, A., Williams, R., and Gilbert, G.* «Gigaplane-XB: Extending the Ultra Enterprise Family», Proc. Hot Interconnects V, IEEE, 1988.

25. *Cody, W.J.* «Analysis of Proposals for the Floating-Point Standard», IEEE Computer Magazine, vol. 14, p. 63–68, Mar. 1981.

26. *Cohen, D.* «On Holy Wars and a Plea for Peace», IEEE Computer Magazine, vol. 14, p. 48–54, Oct. 1981.

27. *Corbaty, F.J.* «PL/1 as a Tool for System Programming», Datamation, vol. 15, p. 68–76, May 1969.

28. *Corbaty, F.J., and Vyssotsky, V. A.* «Introduction and Overview of the MULTICS System», Proc. FJCC, p. 185–196, 1965.

29. *Denning, P.J.* «The Working Set Model for Program Behavior», Commun. of the ACM, vol. 11, p. 323–333, May 1968.

30. *Dijkstra, E. W.* «GOTO Statement Considered Harmful», Commun. of the ACM, vol. 11, p. 147–148, Mar. 1968a.

31. *Dijkstra, E. W.* «Co-operating Sequential Processes», in Programming Languages, F. Genuys (ed.), New York: Academic Press, 1968b.

32. *Driesen, K., and Holzie, URS* «Accurate Indirect Branch Prediction», Proc. 25th Ann. Int'l. Symp. on Computer Arch., ACM, p. 167–177, 1998.

33. *Dubois, M., Scheurich, C., and Briggs, F.A.* «Memory Access Buffering in Multiprocessors», Proc. 13th Ann. Int'l. Symp. on Computer Arch., ACM, p. 434–442, 1986.

34. *Dulong, C.* «The IA-64 Architecture at Work», IEEE Computer Magazine, vol. 31, p. 24–32, July 1998.

35. *Faggin, F., Hoff, M. E.,Jr., Mazor, S., and Shima, M.* «The History of the 4004», IEEE Micro Magazine, vol. 16, p. 10–20, Dec. 1996.

36. *Falsafi, B., and Wood, D.A.* «Reactive NUMA: A Design Unifying S-COMA and CC-NUMA», Proc. 25th Ann. Int'l. Symp. on Computer Arch., ACM, p. 229–240, 1997.

37. *Fisher,J. A., and Freudenberger, S. M.* «Predicting Conditional Branch Directions from Previous Runs of a Program», Proc. 5th Conf. on Arch. Support for Prog. Lang. and Operating Syst., ACM, p. 85–95, 1992.

38. *Floyd, T. L.* Digital Fundamentals, 6th ed., Upper Saddle River, NJ: Prentice Hall, 1997.

39. *Flynn, M.J.* «Some Computer Organizations and Their Effectiveness», IEEE Trans. on Computers, vol. C-21, p. 948–960, Sept. 1972.

40. *Foster, I., and Kesselman, C.* «Globus: A Metacomputing Infrastructure Toolkit», Int'l. J. of Supercomputer Applications, vol. 11, p. 115–128, 1998a.

41. *Foster, I., and Kesselman, C.* «The Globus Project: A Status Report», IPPS/SPDP '98 Heterogeneous Computing Workshop, IEEE, p. 4–18, 1998b.

42. *Fotheringham,J.* «Dynamic Storage Allocation in the Atlas Computer Including an Automatic Use of a Backing Store», Commun. of the ACM, vol. 4, p. 435–436, Oct. 1961.

43. *Gajski, D. D., and Pier, K. -K.* «Essential Issues in Multiprocessor Systems», IEEE Computer Magazine, vol. 18, p. 9–27, June 1985.

44. *Garner, H. L.* «Number Sysytems and Arithmetic», in Advances in Computers, vol. 6, F. Alt and M. Rubinoff (eds.), New York: Academic Press, 1965, p. 131–194.

45. *Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Mancheck, R., and Sunderram, V.* PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing, Cambridge, MA: M.I.T. Press, 1994.

46. *Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J.* «Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors», Proc. 17th Ann. Int'l. Symp. on Computer Arch., ACM, p. 15–26, 1990.

47. *Goodman, J. R.* «Using Cache Memory to Reduce Processor Memory Traffic», Proc. 10th Ann. Int'l. Symp. on Computer Arch., ACM, p. 124–131, 1983.

48. *Goodman J. R.* «Cache Consistency and Sequential Consistency», Tech. Rep. 61, IEEE Scalable Coherent Interface Working Group, IEEE, 1989.

49. *Graham, R.* «Use of High Level Languages for System Programming», Project MAC Report TM-13, Project MAC, MIT, Sept. 1970.

50. *Grimshaw, A. S., and Wulf, W.* «Legion: A View from 50,000 Feet», Proc. Fifth Int'l. Symp. on High-Performance Distributed Computing, IEEE, p. 89–99, Aug. 1996.

51. *Grimshaw, A. S., and Wulf, W.* «The Legion Vision of a Worldwide Virtual Computer», Commun. of the ACM, vol. 40, p. 39–45, Jan. 1997.

52. *Gropp, W., Lusk, E., and Skjellum, A.* «Using MPI: Portable Parallel Programming with the Message Passing Interface», Cambridge, MA: M.I.T. Press, 1994.

53. *Hagersten, E., Landin, A., Haridi, S.* «DDM — A Cache-Only Memory Architecture», IEEE Computer Magazine, vol. 25, p. 44–54, Sept. 1992.

54. *Hamacher, V. V., Vranesic, Z. G., and Zaky, S. G.* Computer Organization, 4th ed., New York: McGraw-Hill, 1996.

55. *Hamming, R. W.* «Error Detecting and Error Correcting Codes», Bell Syst. Tech. J., vol. 29, p. 147–160, April 1950.

56. *Handy, J.* The Cache Memoru Book, 2nd ed., Orlando, FL: Academic Press, 1998.

57. *Hart, J. M.* Win32 System Programming, Reading, MA: Addison-Wesley, 1997.

58. *Hayes, J. P.* Computer Architecture and Organization, 3rd ed., New York: McGraw-Hill, 1998.

59. *Hennessy, J. L.* «VLSI Processor Architecture», IEEE Trans. on Computers, vol. C-33, p. 1221–1246, Dec. 1984.

60. *Hill, M.* «Multiprocessors Should Support Simple Memory-Consistency Models», IEEE Computer Magazine, vol. 31, p. 28–34, Aug. 1998.

61. *Hoare, C. A. R.* «Monitors, An Operating System Structuring Concept», Commun. of the ACM, vol. 17, p. 549–557, Oct. 1974; Erratum in Commun. of the ACM, vol. 18, p. 95, Feb.1975.

62. *Hwang, K., and Xu, Z.* Scalable Parallel Computing, New York: McGraw-Hill, 1998.

63. *Hwu, W. -M.* «Introduction to Predicated Execution», IEEE Computer Magazine, vol. 31, p. 49–50, Jan. 1998.

64. *Irvine, K.* Assembly Language for Intel-Based Computers, 3rd ed., Englewood Cliffs, NJ: Prentice Hall, 1999.

65. *Jacob, B., and Mudge, T.* «Virtual Memory: Issues of Implementation», IEEE Computer Magazine, vol. 31, p. 33–43, June 1998a.

66. *Jacob, B., and Mudge, T.* «Virtual Memory in Contemporary Microprocessors», IEEE Micro Magazine, vol. 18, p. 60–75, July/Aug. 1998b.

67. *Joe, T., and Hennessy, J. L.* «Evaluating the Memory Overhead Required for COMA Architectures», Proc. 21th Ann. Int'l. Symp. on Computer Arch., ACM, p. 82–93, 1994.

68. *Johnson, K. L., Kaashoek, M. F., and Wallach, D. A.* «CRL: High-Performance All-Software Distributed Shared Memory», Proc. 15th Symp. on Operating Systems Principles, ACM, p. 213–228, 1995.

69. *Johnson, M.* Superscalar Microprocessor Design, Englewood Cliffs, NJ: Prentice Hall, 1991.

70. *Juan, T., Sanjeevan, S., and Navarro, J. J.* «Dynamic History-Length Fitting: A Third Level of Adaptivity for Branch Prediction», Proc. 25th Ann. Int'l. Symp. on Computer Arch., ACM. p. 155–166, 1998.

71. *Katayama, Y.* «Trends in Semiconductor Memories», IEEE Micro Magazine, p. 10–17, Nov./Dec. 1997.

72. *Kermarrec., A. -M., Kuz, I., Van Steen, M., and Tanenbaum, A. S.* «A Framework for Consistent Replicated Web Objects», Proc. 18th Int'l. Conf. on Distr. Computing Syst., IEEE, p. 276–284, 1998.

73. *Knuth, D. E.* «An Empirical Study of FORTRAN Programs», Software — Practice & Experience, vol. l, p. 105–133, 1971.

74. *Knuth, D. E.* The Art of Computer Programming: Fundamental Algorithms, 3rd ed.. Reading, MA: Addison-Wesley, 1997.

75. *Knuth, D. E.* The Art of Computer Programming: Seminumerical Algorithms, 3rd ed., Reading, MA: Addison-Wesley, 1998.

76. *Kontothanassis, L., Hunt, G., Stets, R., Hardavellas, N., Cierniad, M., Parthasarathy, S., Meira, W., Dwarkadas, S., and Scott, M.* VM-Based Shared Memory on Low Latency Remote Memory Access Networks, Proc. 24th Ann. Int'l. Symp. on Computer Arch., ACM, p. 157–169, 1997.

77. *Korn, D.* «Porting UNIX to Windows NT», Proc. Winter 1997 USENIX Conf., p. 43–57, 1997.

78. *Kumar, V. P., and Reddy, S. M.* «Augmented Shuffle-Exchange Multistage Interconnection Networks», IEEE Computer Magazine, vol. 20, p. 30–40, June 1987.

79. *Lamport, L.* «How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs», IEEE Trans. on Computers, vol. C-28, p. 690–691, Sept. 1979.

80. *LaRowe, R. P., and Ellis, C. S.* «Experimental Comparison of Memory Management Policies for NUMA Multiprocessors», ACM Trans. on Computer Systems, vol. 9, p. 319–363, Nov. 1991.

81. *Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W. -D., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M.* «The Stanford Dash Multiprocessor», IEEE Computer Magazine, vol. 25, p. 63–79, March 1992.

82. *Li, K.* «IVY: A Shared Virtual Memory System for Parallel Computing», Proc. 1988 Int'l. Conf. on Parallel Proc. (Vol. 11), IEEE, p. 94–101, 1988.

83. *Li, K., and Hudak, P.* «Memory Coherence in Shared Virtual Memory Systems», ACM Trans. on Computer Systems, vol. 7, p. 321–359, Nov. 1989.

84. *Li, K., and Hudak, P.* «Memory Coherence in Shared Virtual Memory Systems», Proc. 5th Ann. ACM Symp. on Prin. of Distr. Computing, ACM, p. 229–239, 1986.

85. *Lindholm, T., and Yellin, F.* The Java Virtual Machine Specification, Reading, MA: Addison-Wesley, 1997.

86. *Loshin, D.* High Performance Computing Demystified, Cambridge, MA: AP Prof., 1994.

87. *Lu, H., Cox, A. L., Dwarkadas, S., Rajamony, R., and Zwaenepoel, W.* «Software Distributed Shared Memory Support for Irregular Applications», Proc. 6th Conf. on Prin. and Practice of Parallel Progr., p. 48–56, June 1997.

88. *Lukasiewicz, J.* Aristotle's Syllogistic, 2nd ed., Oxford: Oxford University Press, 1958.

89. *Mano, M. M., and Kime, C. R.* Logic and Computer Design Fundamentals, Upper Saddle River, NJ: Prentice Hall, 1997.

90. *Martin, R. P., Vahdat, A. M., Culler, D. E., and Anderson, T. E.* «Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture», Proc. 24th Ann. Int'l. Symp. on Computer Arch., ACM, p. 85–97, 1997.

91. *Mazidi, M. A., and Mazidi, J. G.* The 80x86 IBM PC and Compatible Computers, 2nd ed., Upper Saddle River, NJ: Prentice Hall, 1998.

92. *McGhan, H. and O'connor, J. M.* «picoJava: A Direct Execution Engine for Java Bytecode», IEEE Computer Magazine, vol. 31., Oct. 1998.

93. *McKee, S. A., Klenke, R. H., Wright, K. L., Wulf, W. A., Saunas, M. H., Aylor, J. H., and Batson, A. P.* «Smarter Memory: Improving Bandwidth for Streamed References», IEEE Computer Magazine, vol. 31, p. 54–63, July 1998.

94. *McKusick, M. K., Bostic, K., Karels, M., and Quarterman, J. S.* «The Design and Implementation of the 4.4 BSD Operating System», Reading, MA: Addison-Wesley, 1996.

95. *McKusick, M. K., Joy, W. N., Leffler, S. J., and Fabry, R. S.* «A Fast File System for UNIX», ACM Trans. on Computer Systems, vol. 2, p. 181–197, Aug. 1984.

96. *Messmer, H.-P.* The Indispensible PC Hardware Book, 3rd ed.. Reading, MA: Addison-Wesley, 1997.

97. *Morgan, C.* Portraits in Computing, New York: ACM Press, 1997.

98. *Morin, C., Gefflaut, A., Banatre, M., and Kermarrec, A. -M.* «COMA: An Opportunity for Building a Fault-Tolerant Scalable Shared Memory Multiprocessor», Proc. 24th Ann. tnt'l. Symp. on Computer Arch., ACM, p. 65–65, 1996.

99. *Moudgill, M., and Vassiliadis, S.* «Precise Interrupts», IEEE Micro Magazine, vol. 16, p. 58–67, Feb. 1996.

100. *Mullender, S. J., and Tanenbaum, A. S.* «Immediate Files», Software — Practice and Experience, vol. 14, p. 365–368, 1984.

101. *Nelson, V. P., Nagle, H. T., Carroll, B. D., and Irwin, J. D.* Digital Logic and Circuit Analysis and Design, Englewood Cliffs, NJ: Prentice Hall, 1995.

102. *Ng, S. W.* «Advances in Disk Technology: Performance Issues», IEEE Computer Magazine, vol. 31, p. 75–81, May 1998.

103. *Normoyle, K. B., Csoppenszky, M. A., Tzeng, A., Johnson, T. P., Furman, C. D., and Mos-Toufi, J.* «UltraSPARC Hi: Expanding the Boundaries of a System on a Chip», IEEE Micro Magazine, vol. 18, p. 14–24, March/April 1998.

104. *Norton, P., and Goodman, J.* Inside the PC, 7th ed., Indianapolis, IN: Sams, 1997.

105. *O'connor, J.M., and Tremblay, M.* «PicoJava-I: The Java Virtual Machine in Hardware», IEEE Micro Magazine, vol. 17, p. 45–53, March/April 1997.

106. *Organick, E.* The MULTICS System, Cambridge, MA: M.I.T. Press, 1972.

107. *Pakin, S., Karamcheti, V., and Cfflen, A. A.* «Fast Messages (FM): Efficient, Portable Communication for Workstation Cluster and Massively-Parallel Processors», IEEE Concurrency, vol. 5, p. 60–73, April-June 1997.

108. *Pan, S. -T., So, K., and Rahmeh, J. T.* «Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation», Proc. 5th Int'l. Conf. on Arch. Support for Prog. Long. and Operating Syst., ACM, p. 76–84, Oct. 1992.

109. *Papamarcos, M., and Patel., J.* «A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories», Proc. 11th Ann. Int'l. Symp. on Computer Arch., ACM, p. 348–354, 1984.

110. *Patterson, D. A.* «Reduced Instruction Set Computers», Commun. of the ACM, vol. 28, p. 8–21, Jan. 1985.

111. *Patterson, D. A., Gibson, G., and Katz, R.* «A case for redundant arrays of inexpensive disks (RAID)», Proc. ACM SIGMOD Int'l. Conf. on Management of Data, ACM, p. 109–166, 1988.

112. *Patterson, D. A., and Hennessy, J. L.* Computer Organization and Design, 2nd ed., San Francisco, CA: Morgan Kaufmann, 1998.

113. *Patterson, D. A., and Sŭquin, C. H.* «A VLSI RISC», IEEE Computer Magazine, vol. 15, p. 8–22, Sept. 1982.

114. *Paul, R. P.* SPARC Architecture, Assembly Language, Programming, and C, Englewood Cliffs, NJ: Prentice Hall, 1994.

115. *Pfister, G. F.* In Search of Clusters, 2nd ed., Upper Saddle River, NJ: Prentice Hall, 1998.

116. *Pilgrim, A.* Build Your Own Pentium II PC, New York: McGraw-Hill, 1998.

117. *Pountain, D.* «Pentium: More RISC than CISC», Byte, vol. 18, p. 195–204, Sept. 1993.

118. *Price, D.* «A History of Calculating Machines», IEEE Micro Magazine, vol. 4, p. 22–52, Feb.1984.

119. *Radin, G.* «The 801 Minicomputer», Computer Arch. News, vol. 10, p. 39–47, March 1982.

120. *Ritchie, D. M., and Thompson, K.* «The UNIX Time-Sharing System», Commun. of the ACM, vol. 17, p. 365–375, July 1974.

121. *Rosenblum, M., and Ousterhout, J. K.* «The Design and Implementation of a Log-Structured File System», Proc. Thirteenth Symp. on Operating System Principles, ACM, p. 1–15, 1991.

122. *Saloman, D.* Assemblers and Loaders, Englewood Cliffs, NJ: Prentice Hall, 1993.

123. *Saulsbury, A., Wilkinson, T., Carger, J., and Landin, A.* «An Argument for Simple COMA», Proc. of First IEEE Symp. on High-Performance Comp. Arch., IEEE, p. 276–285, 1995.

124. *Scales, D. J., Gharachorloo, K., and Thekkath, CA.* «Shasta: A Low-Overhead Software-Only Approach for Supporting Fine-Grain Shared Memory», Proc. 7th Int'l. Conf. on Arch. Support for Prog. Long. and Oper. Syst., ACM, p. 174–185, 1996.

125. *Sechrest, S., Lee, C. -C., and Mudge, T.* «Correlation and Aliasing in Dynamic Branch Predictors», Proc. 23th Ann. Int'l. Symp. on Computer Arch., ACM, p. 22–32, 1996.

126. *Seltzer, M., Bostic, K., McKusick, M.K., and Staelin, C.* «An Implementation of a Log-Structured File System for UNIX», Proc. Winter 1993 USENIX Technical Conf., p. 307–326, 1993.

127. *Shanley, T., and Anderson, D.* ISA System Architecture, Reading, MA: Addison-Wesley, 1995a.

128. *Shanley, T., and Anderson, D.* PCI System Architecture, 3rd ed.. Reading, MA: Addison-Wesley, 1995b.

129. *Shriver, B., and Smith, B.* The Anatomy of a High-Performance Microprocessor: A Systems Perspective, Los Alamitos, CA: IEEE Computer Society, 1998.

130. *Sima, D.* «Superscalar Instruction Issue», IEEE Micro Magazine, vol. 17, p. 28–39, Sept./Oct 1997.

131. *Sima, D., Fountain, T., and Kacsuk, P.* Advanced Computer Architectures: A Design Space Approach, Reading, MA: Addison-Wesley, 1997.

132. *Slater, R.* Portraits in Silicon, Cambridge, MA: M.I.T. Press, 1987.

133. *Smith, A. J.* «Cache Memories», Computing Surveys, vol. 14, p. 473–530, Sept. 1982.

134. *Snip, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., and Dongarra, J.* MPI: The Complete Reference Manual, Cambridge, MA: M.I.T. Press, 1996.

135. *Solari, E.* ISA & EISA Theory and Operation, San Diego, CA: Annabooks, 1993.

136. *Solari, E., and Willse, G.* PCI Hardware and Software Architecture and Design, 4th ed., San Diego, CA: Annabooks, 1998.

137. *Solomon, D.A.* Inside Windows NT, 2nd ed., Redmond, WA: Microsoft Press, 1998.

138. *Sprangle, E., Chappell, R. S., Alsup, M., and Patt, Y. N.* «The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference», Proc. 24th Ann. Int'l. Symp. on Computer Arch., ACM, p. 284–291, 1997.

139. *Stallings, W.* Computer Organization and Architecture, 4th ed. Upper Saddle River, NJ: Prentice Hall, 1996.

140. *Stenstrom, P., Hagersten, E., Lilja, D. J., Martonosi, M., and Venugopal, M.* «Trends in Shared Memory Multiprocessing», IEEE Computer Magazine, vol. 30, p. 44–50, Dec. 1997.

141. *Stets, R., Dwarkadas, S., Hardavellas, N., Hunt, G., Kontothanassis, L., Parthasarathy, S., and Scott, M.* «CASHMERE-2L: Software Coherent Shared Memory on Clustered Remote-Write Networks», Proc. 16th Symp. on Operating Systems Principles, ACM, p. 170–183, 1997.

142. *Sunderram, V. B.* «PVM: A Framework for Parallel Distributed Computing», Concurrency: Practice and Experience, vol. 2, p. 315–339, Dec. 1990.

143. *Swan, R. J., Fuller, S. H., and Siewiorek, D. P.* «Cm* —A Modular Multiprocessor», Proc. NCC, p. 645–655, 1977.

144. *Tan, W. M.* Developing USB PC Peripherals, San Diego, CA: Annabooks, 1997.

145. *Tanenbaum, A. S.* «Implications of Structured Programming for Machine Architecture», Commun. of the ACM, vol. 21, p. 237–246, Mar. 1978.

146. *Tanenbaum, A. S., and Woodhull, A. W.* Operating Systems: Design and Implementation, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1997.

147. *Thompson, K.* «UNIX Implementation», Bell Syst. Tech. J., vol. 57, p. 1931–1946, July-Aug. 1978.

148. *Treleaven, P.* «Control-Driven, Data-Driven, and Demand-Driven Computer Architecture», Parallel Computing, vol. 2, 1985.

149. *Tremblay, M. and O'connor, J. M.* «UltraSPARC I: A Four-Issue Processor Supporting Multimedia», IEEE Micro Magazine, vol. 16, p. 42–50, April 1996.

150. *Triebel, W. A.* The 80386, 80486, and Pentium Processor, Upper Saddle River, NJ: Prentice Hall, 1998.

151. *Unger, S. H.* «A Computer Oriented Toward Spatial Problems», Proc. IRE, vol. 46, p. 1744–1750, 1958.

152. *Vahalia, U.* UNIX Internals, Upper Saddle River, NJ: Prentice Hall, 1996.

153. *Van Der Poel, W. L.* «The Software Crisis, Some Thoughts and Outlooks», Proc. IFIP Congr. 68, p. 334–339, 1968.

154. *Van Steen, M., Homburg, P. C., and Tanenbaum, A. S.* «The Architectural Design of Globe: A Wide-Area Distributed System», IEEE Concurrency, vol. 7, p. 70–78, Jan.-March 1999.

155. *Verstoep, K., Langedoen, K., and Bal, H. E.* «Efficient Reliable Mulicast on Myrinet», Proc. 1996 Int'l. Conf. on Parallel Processing, IEEE, p. 156–165, 1996.

156. *Weaver, D. L., and Germond, T.* The SPARC Architecture Manual, Version 9, Englewood Cliffs, NJ: Prentice Hall, 1994.

157. *Wilkes, M. V.* «Computers Then and Now», J. ACM, vol. 15, p. 1–7, Jan. 1968.

158. *Wilkes, M. V.* «The Best Way to Design an Automatic Calculating Machine», Proc. Manchester Univ. Computer Inaugural Conf., 1951.

159. *Wilkinson, B.* Computer Architecture: Design and Performance, 2nd ed., Englewood Cliffs, NJ: Prentice Hall, 1994. Wilinson, 1994.

160. *Wilkinson, B. and Allen, M.* Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, Upper Saddle River, NJ: Prentice Hall, 1999.

161. *Wilson, J.* «Challenges and Trends in Processor Design», IEEE Computer Magazine, vol. 31, p. 39–48, Jan. 1998.

162. *Wilson, P.* «Floating-Point Survival Kit», Byte, vol. 13, p. 217–226, March 1988.

163. *Yeh, T.-Y., and Patt, Y. -N.* «Two-Level Adaptive Training Branch Prediction», Proc. 24th Int'l. Symp. on Microarchitectwe, ACM/IEEE, p. 51–61,1991.

Приложение А
Двоичные числа

Арифметика, применяемая в компьютерах, сильно отличается от арифметики, которая используется людьми. Во-первых, компьютеры выполняют операции над числами, точность которых конечна и фиксирована. Во-вторых, в большинстве компьютеров используется не десятичная, а двоичная система счисления. Этим двум проблемам и посвящено приложение.

Числа конечной точности

Когда люди выполняют какие-либо арифметические действия, их не волнует вопрос, сколько десятичных разрядов занимает то или иное число. Физики, к примеру, могут вычислить, что во Вселенной присутствует 10^{78} электронов, и их не волнует тот факт, что полная запись этого числа потребует 79 десятичных разрядов. Никогда не возникает проблемы нехватки бумаги для записи числа.

С компьютерами дело обстоит иначе. В большинстве компьютеров количество доступной памяти для хранения чисел фиксировано и зависит от того, когда был разработан этот компьютер. Если приложить усилия, программист сможет представлять числа в два, три и более раз большие, чем позволяет размер памяти, но это не меняет природы данной проблемы. Память компьютера ограничена, поэтому мы можем иметь дело только с такими числами, которые можно представить в фиксированном количестве разрядов. Такие числа называются **числами конечной точности**.

Рассмотрим ряд положительных целых чисел, которые можно записать тремя десятичными разрядами без десятичной запятой и без знака. В этот ряд входит ровно 1000 чисел: 000, 001, 002, 003, ..., 999. При таком ограничении невозможно выразить определенные типы чисел. Сюда входят:

1. Числа больше 999.
2. Отрицательные числа.
3. Дроби.
4. Иррациональные числа.

5. Комплексные числа.

Одно из свойств набора всех целых чисел — **замкнутость** по отношению к операциям сложения, вычитания и умножения. Другими словами, для каждой пары целых чисел i и j числа $i+j$, $i-j$ и $i \times j$ — тоже целые числа. Ряд целых чисел не замкнут относительно деления, поскольку существуют такие значения i и j, для которых i/j не выражается в виде целого числа (например, 7/2 или 1/0).

Числа конечной точности не замкнуты относительно всех четырех операций. Ниже приведены примеры операций над трехразрядными десятичными числами:

600+600=1200 (слишком большое число);
003–005=–2 (отрицательное число);
050×050=2500 (слишком большое число);
007/002=3,5 (не целое число).

Отклонения можно разделить на два класса: операции, результат которых превышает самое большое число ряда (ошибка переполнения) или меньше, чем самое маленькое число ряда (ошибка из-за потери значимости), и операции, результат которых не является слишком маленьким или слишком большим, а просто не является членом ряда. Из четырех примеров, приведенных выше, первые три относятся к первому классу, а четвертый — ко второму классу.

Поскольку размер памяти компьютера ограничен и он должен выполнять арифметические действия над числами конечной точности, результаты определенных вычислений будут неправильными с точки зрения классической математики. Вычислительное устройство, которое выдает неправильный ответ, может показаться странным на первый взгляд, но ошибка в данном случае — это только следствие его конечной природы. Некоторые компьютеры содержат специальное аппаратное обеспечение, которое обнаруживает ошибки переполнения.

Алгебра чисел конечной точности отличается от обычной алгебры. В качестве примера рассмотрим ассоциативный закон

$$a+(b-c)=(a+b)-c.$$

Вычислим обе части выражения для a=700, b=400 и c=300. В левой части сначала вычислим значение $(b-c)$. Оно равно 100. Затем прибавим это число к a и получим 800. Чтобы вычислить правую часть, сначала вычислим $(a+b)$.

Для трехразрядных целых чисел получится переполнение. Результат будет зависеть от компьютера, но он не будет равен 1100. Вычитание 300 из какого-то числа, отличного от 1100, не даст результата 800. Ассоциативный закон не имеет силы. Порядок операций важен.

Другой пример — дистрибутивный закон:

$$a \times (b-c)=a \times b-a \times c.$$

Сосчитаем обе части выражения для a=5, b=210 и c=195. В левой части 5×15=75. В правой части 75 не получается, поскольку $a \times b$ выходит за пределы ряда.

Исходя из этих примеров, кто-то может сделать вывод, что компьютеры совершенно непригодны для выполнения арифметических действий. Вывод, естествен-

но, неверен, но эти примеры наглядно иллюстрируют важность понимания, как работает компьютер и какие ограничения он имеет.

Позиционные системы счисления

Обычное десятичное число состоит из цепочки десятичных разрядов и иногда десятичной запятой. Общая форма записи показана на рис. А.1. Десятка выбрана в качестве основы возведения в степень (это называется **основанием системы счисления**), поскольку мы используем 10 цифр. В компьютерах удобнее иметь дело с другими основаниями системы счисления. Самые важные из них — 2, 8 и 16. Соответствующие системы счисления называются **двоичной**, **восьмеричной** и **шестнадцатеричной** соответственно.

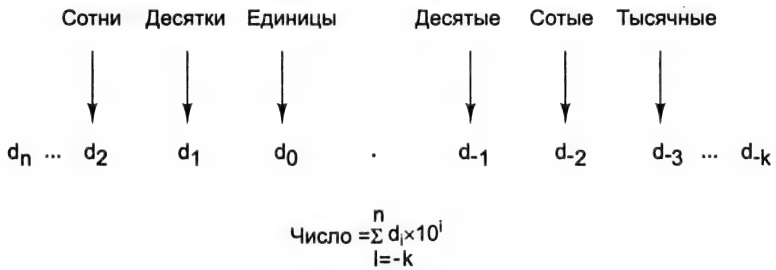


Рис. А.1. Общая форма десятичного числа

k-ичная система требует k различных символов для записи разрядов с 0 по $k-1$. Десятичные числа строятся из 10 десятичных цифр

0 1 2 3 4 5 6 7 8 9

Двоичные числа, напротив, строятся только из двух двоичных цифр

0 1

Восьмеричные числа состоят из восьми цифр

0 1 2 3 4 5 6 7

Для шестнадцатеричных чисел требуется 16 цифр. Это значит, что нам нужно 6 новых символов. Для обозначения цифр, следующих за 9, принято использовать прописные латинские буквы от A до F. Таким образом, шестнадцатеричные числа строятся из следующих цифр.

0 1 2 3 4 5 6 7 8 9 A B C D E F

Двоичный разряд (то есть 1 или 0) обычно называют **битом**. На рис. А.2 десятичное число 2001 представлено в двоичной, восьмеричной и шестнадцатеричной системе. Число 7B9 очевидно шестнадцатеричное, поскольку символ B встречается только в шестнадцатеричных числах. А число 111 может быть в любой из четырех систем счисления. Чтобы избежать двусмысленности, нужно использовать индекс для указания основания системы счисления.

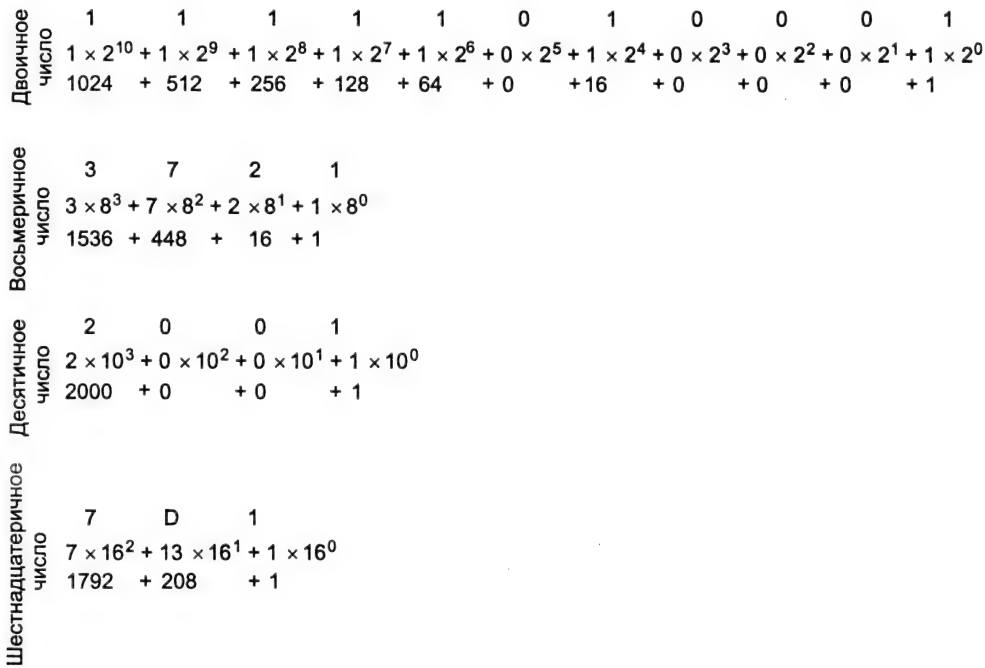


Рис. А.2. Число 2001 в двоичной, восьмеричной и шестнадцатеричной системе

В таблице А.1 ряд неотрицательных целых чисел представлен в каждой из четырех систем счисления.

Таблица А.1. Десятичные числа и их двоичные, восьмеричные и шестнадцатеричные эквиваленты

Десятичное	Двоичное	Восьмеричное	Шестнадцатеричное
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D

Десятичное	Двоичное	Восьмеричное	Шестнадцатеричное
14	1110	16	E
15	1111	17	F
16	10000	20	10
20	10100	24	14
30	11110	36	1E
40	101000	50	28
50	110010	62	32
60	111100	74	3C
70	1000110	106	46
80	1010000	120	50
90	1011010	132	5A
100	11001000	144	64
1000	1111101000	1750	3E8
2989	101110101101	5655	BAD

Преобразование чисел из одной системы счисления в другую

Преобразовывать числа из восьмеричной в шестнадцатеричную или двоичную систему и обратно легко. Чтобы преобразовать двоичное число в восьмеричное, нужно разделить его на группы по три бита, причем три бита непосредственно слева от двоичной запятой формируют одну группу, следующие три бита слева от этой группы формируют вторую группу и т. д. Каждую группу по три бита можно преобразовать в один восьмеричный разряд со значением от 0 до 7 (см. первые строки табл. А.1). Чтобы дополнить группу до трех битов, нужно спереди приписать один или два нуля. Преобразование из восьмеричной системы в двоичную тоже тривиально. Каждый восьмеричный разряд просто заменяется эквивалентным 3-битным числом. Преобразование из 16-ричной в двоичную систему, по сути, сходно с преобразованием из 8-ричной в двоичную систему, только каждый 16-ричный разряд соответствует группе из четырех битов, а не из трех. На рис. А.3 приведены примеры преобразований из одной системы в другую.

Преобразование десятичных чисел в двоичные можно совершать двумя разными способами. Первый способ непосредственно вытекает из определения двоичных чисел. Самая большая степень двойки, меньшая, чем число, вычитается из этого числа. Та же операция проделывается с полученной разностью. Когда число разложено по степеням двойки, двоичное число может быть получено следующим образом. Единички ставятся в тех позициях, которые соответствуют полученным степеням двойки, а нули — во всех остальных позициях.

Второй способ — деление числа на 2. Частное записывается непосредственно под исходным числом, а остаток (0 или 1) записывается рядом с частным. То же

проделывается с полученным частным. Процесс повторяется до тех пор, пока не останется 0. В результате должны получиться две колонки чисел — частных и остатков. Двоичное число можно считать из колонки остатков снизу вверх. На рисунке А.4 показано, как происходит преобразование из десятичной в двоичную систему.

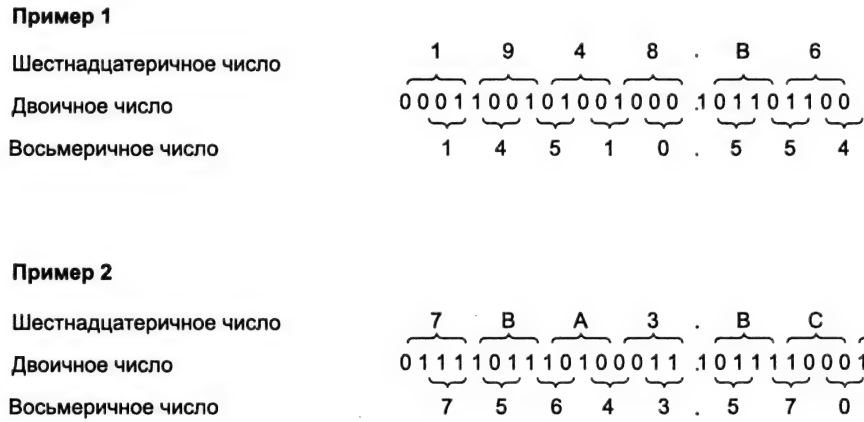


Рис. А.3. Примеры преобразования из 8-ричной системы счисления в двоичную и из 16-ричной в двоичную

Двоичные числа можно преобразовывать в десятичные двумя способами. Первый способ — суммирование степеней двойки, которые соответствуют биту 1 в двоичном числе. Например:

$10110=2^4+2^2+2^1=16+4+2=22$

Второй способ. Двоичное число записывается вертикально по одному биту в строке, крайний левый бит находится внизу. Самая нижняя строка — это строка 1, затем идет строка 2 и т. д. Десятичное число строится напротив этой колонки. Сначала обозначим строку 1. Элемент строки n состоит из удвоенного элемента строки $n-1$ плюс бит строки n (0 или 1). Элемент, полученный в самой верхней строке, и будет ответом. Метод проиллюстрирован на рис. А.5.

Преобразование из десятичной в восьмеричную или 16-ричную систему можно выполнить либо путем преобразования сначала в двоичную, а затем в нужную нам систему, либо путем вычитания степеней 8 или 16.

Отрицательные двоичные числа

На протяжении всей истории цифровых компьютеров для репрезентации отрицательных чисел использовались 4 различные системы. Первая из них называется системой **со знаком**. В такой системе крайний левый бит — это знаковый бит (0 — это «+», а 1 — это «–»), а оставшиеся биты показывают абсолютное значение числа.

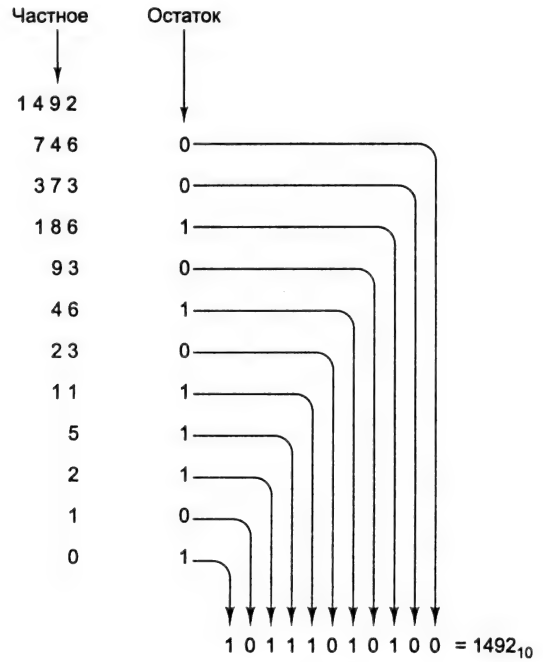


Рис. А.4. Преобразование десятичного числа 1492 в двоичное путем последовательного деления (сверху вниз). Например, 93 делится на 2, получается 46 и остаток 1. Остаток записывается в строку внизу

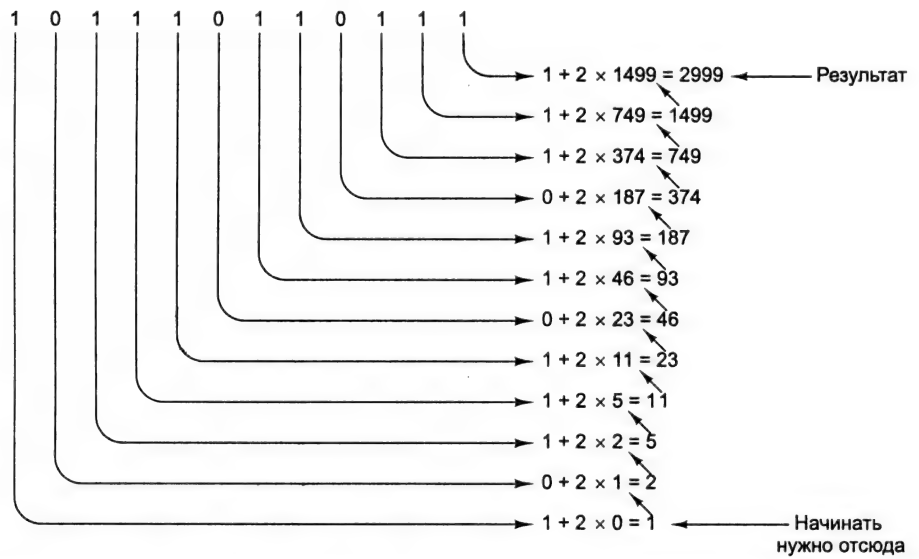


Рис. А.5. Преобразование двоичного числа 101110110111 в десятичное путем последовательного удваивания снизу вверх. В каждой следующей строке удваивается значение предыдущей строки и прибавляется соответствующий бит. Например, 374 умножается на 2 и прибавляется бит соответствующей строки (в данном случае 1). В результате получается 749

Во второй системе, которая называется **дополнением до единицы**, тоже присутствует знаковый бит (0 — это плюс, а 1 — это минус). Чтобы сделать число отрицательным, нужно заменить каждую 1 на 0 и каждый 0 на 1. Это относится и к знаковому биту. Система дополнения до единицы уже устарела.

Третья система, **дополнение до двух**, содержит знаковый бит (0 — это «+», а 1 — это «–»). Отрицание числа происходит в два этапа. Сначала каждая единица меняется на 0, а каждый 0 — на 1 (как и в системе дополнения до единицы). Затем к полученному результату прибавляется 1. Двоичное сложение происходит точно так же, как и десятичное, только перенос совершается в том случае, если сумма больше 1, а не больше 9. Например, рассмотрим преобразование числа 6 в форму с дополнением до двух:

00000110 (+6);
11111001 (–6 в системе с дополнением до единицы);
11111010 (–6 в системе с дополнением до двух).

Если нужно совершить перенос из крайнего левого бита, он просто отбрасывается.

В четвертой системе, которая для m-битных чисел называется **excess 2^{m-1}**, число представляется как сумма этого числа и 2^{m-1}. Например, для 8-битного числа (m=8) система называется excess 128, а число сохраняется в виде суммы исходного числа и 128. Следовательно, –3 превращается в –3+128=125, и это число (–3) представляется 8-битным двоичным числом 125 (01111101). Числа от –128 до +127 выражаются числами от 0 до 255 (все их можно записать в виде 8-битного положительного числа). Отметим, что эта система соответствует системе с дополнением до двух с обращенным знаковым битом. В таблице А.2 представлены примеры отрицательных чисел во всех четырех системах.

Таблица А.2. Отрицательные 8-битные числа в четырех различных системах

N десятичное	N двоичное	–N в системе со знаком	–N дополнение до единицы	–N дополнение до двух	–N excess 128
1	00000001	10000001	11111110	11111111	011111111
2	00000010	10000010	11111101	11111110	01111110
3	00000011	10000011	11111100	11111101	01111101
4	00000100	10000100	11111011	11111100	01111100
5	00000101	10000101	11111010	11111011	01111011
6	00000110	10000110	11111001	11111010	01111010
7	00000111	10000111	11111000	11111001	01111001
8	00001000	10001000	11110111	11111000	01111000
9	00001001	10001001	11110110	11110111	01110111
10	00001010	10001010	11110101	11110110	01110110
20	00010100	10010100	11101011	11101100	01101100
30	00011110	10011110	11100001	11100010	01100010
40	10101000	10101000	11010111	11011000	01011000
50	00110010	10110010	11001101	11001110	01001110
60	00111100	10111100	11000011	11000100	01000100
70	01000110	11000110	10111001	10111010	00111010
80	01010000	11010000	10101111	10110000	00110000
90	01011010	11011010	10100101	10100110	00100110
100	01100100	11100100	10011011	10011100	00011100
127	01111111	11111111	10000000	10000001	00000001
128	Не сущест.	Не сущест.	Не сущест.	10000000	00000000

В системах со знаком и с дополнением до единицы есть два представления нуля: +0 и −0. Такая ситуация нежелательна. В системе с дополнением до двух такой проблемы нет, поскольку здесь плюс нуль это всегда плюс нуль. Но зато в этой системе есть другая особенность. Набор битов, состоящий из 1, за которой следуют все нули, является дополнением самого себя. В результате ряд положительных и отрицательных чисел несимметричен — существует одно отрицательное число без соответствующего ему положительного.

Мы считаем это проблемами, поскольку хотим иметь систему кодировки, в которой:

1. Существует только одно представление нуля.
2. Количество положительных чисел равно количеству отрицательных.

Дело в том, что любой ряд чисел с равным количеством положительных и отрицательных чисел и только одним нулем содержит нечетное число членов, тогда как *m* битов предполагают четное число битовых комбинаций. В любом случае будет либо одна лишняя битовая комбинация, либо одной комбинации не будет доставать. Лишнюю битовую комбинацию можно использовать для обозначения числа −0, для большого отрицательного числа или для чего-нибудь еще, но она всегда будет создавать неудобства.

Двоичная арифметика

Ниже приведена таблица сложения для двоичных чисел (рис. А.6).

Первое слогаемое	0	0	1	1
Второе слогаемое	+0	+1	+0	+1
Сумма	0	1	1	0
Перенос	0	0	0	1

Рис. А.6. Таблица сложения для двоичных чисел

Сложение двух двоичных чисел начинается с крайнего правого бита. Суммируются соответствующие биты в первом и втором слагаемом. Перенос совершается на одну позицию влево, как и в десятичной арифметике. В арифметике с дополнением до единицы перенос от сложения крайних левых битов прибавляется к крайнему правому биту. Этот процесс называется циклическим переносом. В арифметике с дополнением до двух перенос, полученный в результате сложения крайних левых битов, просто отбрасывается. Примеры арифметических действий над двоичными числами показаны на рис. А.7.

Если первое и второе слагаемые имеют противоположные знаки, ошибки переполнения не произойдет. Если они имеют одинаковые знаки, а результат — противоположный знак, значит, произошла ошибка переполнения и результат неверен. И в арифметике с дополнением до единицы, и в арифметике с дополнением до двух переполнение происходит тогда и только тогда, когда перенос в знаковый бит

отличается от переноса из знакового бита. В большинстве компьютеров перенос из знакового бита сохраняется, но перенос в знаковый бит не виден из ответа, поэтому обычно вводится специальный бит переполнения.

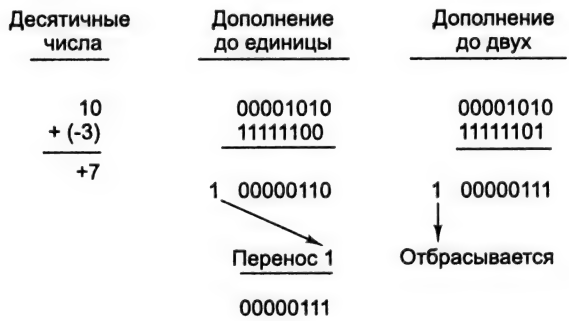


Рис. А.7. Сложение в системах с дополнением до единицы и с дополнением до двух

Вопросы и задания

1. Преобразуйте следующие числа в двоичные: 1984, 4000, 8192.
2. Преобразуйте двоичное число 1001101001 в десятичную, восьмеричную и шестнадцатеричную системы.
3. Какие из следующих цепочек символов являются шестнадцатеричными числами? BED, CAB, DEAD, DECADE, ACCEDED, BAG, DAD.
4. Выразите десятичное число 100 в системах счисления с основаниями от 2 до 9.
5. Сколько различных положительных целых чисел можно выразить в k разрядах, используя числа с основанием системы счисления r?
6. Большинство людей с помощью пальцев на руках могут сосчитать до 10. Однако компьютерщики способны на большее. Представим, что каждый палец соответствует одному двоичному разряду. Пусть вытянутый палец означает 1, а загнутый — 0. До скольки мы можем сосчитать, используя пальцы обеих рук? А если рассматривать пальцы на руках и на ногах? Представим, что большой палец левой ноги — это знаковый бит для чисел с дополнением до двух. Сколько чисел можно выразить таким способом?
7. Выполните следующие вычисления над 8-битными числами с дополнением до двух:

00101101	11111111	00000000	11110111
+01101111	+11111111	−11111111	−11110111

8. Выполните те же вычисления в системе с дополнением до единицы.
9. Ниже приведены задачи на сложение 3-битных двоичных чисел в системе с дополнением до двух. Для каждой суммы установите:
   а. Равен ли знаковый бит результата 1.
   б. Равны ли младшие три бита 0.

в. Не произошло ли переполнения.

000	000	111	100	100
+001	+111	+110	+111	+100

10. Десятичные числа со знаком, состоящие из n разрядов, можно представить в $n+1$ разрядах без знака. Положительные числа содержат 0 в крайнем левом разряде. Отрицательные числа получаются путем вычитания каждого разряда из 9. Например, отрицательным числом от 014725 будет 985274. Такие числа называются числами с дополнением до девяти. Они аналогичны двоичным числам с дополнением до единицы. Выразите следующие числа в виде 3-разрядных чисел в системе с дополнением до девяти: 6, –2, 100, –14, –1, 0.
11. Сформулируйте правило для сложения чисел с дополнением до девяти, а затем выполните следующие вычисления:

0001	0001	9997	9241
+9999	+9998	+9996	+0802

12. Система с дополнением до десяти аналогична системе с дополнением до двух. Отрицательное число в системе с дополнением до десяти получается путем прибавления 1 к соответствующему числу с дополнением до девяти без учета переноса. По какому правилу происходит сложение в системе с дополнением до десяти?
13. Составьте таблицы умножения для чисел системы счисления с основанием 3.
14. Перемножьте двоичные числа 0111 и 0011.
15. Напишите программу, которая на входе получает десятичное число со знаком в виде цепочки ASCII, а на выходе выводит представление этого числа в восьмеричной, шестнадцатеричной и двоичной системе с дополнением до двух.
16. Напишите программу, которая на входе получает 2 цепочки из 32 символов ASCII, содержащие нули и единицы. Каждая цепочка представляет 32-битное двоичное число в системе с дополнением до двух. На выходе программа должна выдавать их сумму в виде цепочки из 32 символов ASCII, содержащей нули и единицы.

Приложение Б
Числа с плавающей точкой

Диапазон чисел, используемых при различных вычислениях, очень велик. Например, в астрономические вычисления может включаться масса электрона (9×10^{-28} граммов) и масса Солнца (2×10^{33} граммов). Диапазон чисел здесь превышает 10^{60}. Эти числа можно представить следующим образом:

```
000000000000000000000000000000000.00000000000000000000000000009
200000000000000000000000000000000.00000000000000000000000000000
```

При всех вычислениях должны сохраняться 34 разряда слева от десятичной запятой и 28 разрядов справа от нее. Это даст 62 значимых разряда в результатах. На бинарном компьютере можно использовать арифметику с многократно увеличенной точностью, чтобы обеспечить достаточную значимость. Однако мы не можем определить массу Солнца с точностью даже до пяти значимых разрядов, не говоря уже о 62. В действительности практически невозможно выполнить какие-либо измерения с точностью до 62 знаков. Можно было бы хранить все промежуточные результаты с точностью до 62 значимых разрядов, а перед выводом окончательных результатов отбрасывать 50 или 60 разрядов, но процессор и память тратили бы на это слишком много времени.

Нам нужна такая система для представления чисел, в которой диапазон выражаемых чисел не зависит от числа значимых разрядов. В этом приложении мы расскажем о такой системе. В ее основе лежит экспоненциальное представление чисел, которое применяется в физике, химии и машиностроении.

Принципы представления с плавающей точкой

Числа можно выражать в следующей общепринятой экспоненциальной форме:

$$n=f\times10^{e}$$

где f называется **дробью,** или **мантиссой**, а e (это положительное или отрицательное целое число) называется **экспонентой**. Компьютерная версия такого представления называется **представлением с плавающей точкой**. Ниже приведены примеры чисел в такой записи.

$3{,}14 = 0{,}314\times10^{1}=3{,}14\times10^{0}$
$0{,}000001 = 0{,}1\times10^{-5}=1{,}0\times10^{-6}$
$1941 = 0{,}1941\times10^{4}=1{,}941\times10^{3}$

Область значений определятся по числу разрядов в экспоненте, а точность определяется по числу разрядов в мантиссе. Существует несколько способов представления того или иного числа, поэтому одна форма выбирается в качестве стандартной. Чтобы изучить свойства такого способа представления, рассмотрим представление R с трехразрядной мантиссой со знаком в диапазоне $0{,}1 \le |f| < 1$ и двухразрядной экспонентой со знаком. Эти числа находятся в диапазоне от $+0{,}100\times10^{-99}$ до $+0{,}999\times10^{+99}$, то есть простираются почти на 199 значимых разрядов, хотя для записи числа требуется всего 5 разрядов и 2 знака.

Числа с плавающей точкой можно использовать для моделирования системы действительных чисел в математике, хотя здесь есть несколько существенных различий. На рис. Б.1 представлена ось действительных чисел. Она разбита на 7 областей:

1. Отрицательные числа меньше $-0{,}999\times10^{99}$.
2. Отрицательные числа от $-0{,}999\times10^{99}$ до $-0{,}100\times10^{-99}$.
3. Отрицательные числа от $-0{,}100\times10^{-99}$ до нуля.
4. Нуль.
5. Положительные числа от 0 до $0{,}100\times10^{-99}$.
6. Положительные числа от $0{,}100\times10^{-99}$ до $0{,}999\times10^{99}$.
7. Положительные числа больше $0{,}999\times10^{99}$.

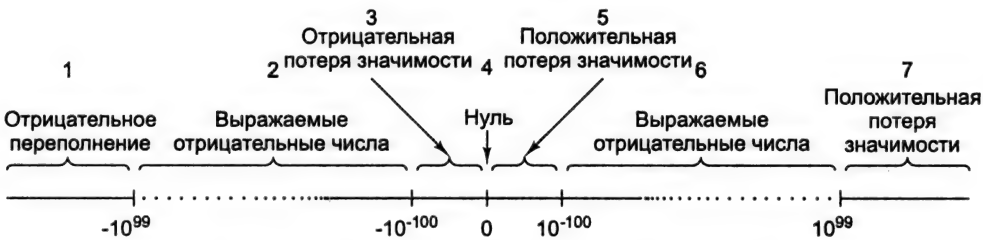


Рис. Б.1. Ось действительных чисел разбита на 7 областей

Первое отличие действительных чисел от чисел с плавающей точкой, которые записываются тремя разрядами в мантиссе и двумя разрядами в экспоненте, состоит в том, что последние нельзя использовать для записи чисел из областей 1, 3, 5 и 7. Если в результате арифметической операции получится число из области 1 или 7 (например, $10^{60}\times10^{60}=10^{120}$), то произойдет **ошибка переполнения** и результат будет неверным. Причина — ограничение области значений чисел в данном представлении. Точно так же нельзя выразить результат из области 3 или 5. Такая ситуация называется **ошибкой из-за потери значимости**. Эта ошибка менее серьезна, чем ошибка переполнения, поскольку часто нуль является вполне удовлетворительным приближением для чисел из областей 3 или 5. Остаток счета в банке на 10^{-102} не сильно отличается от остатка счета 0.

Второе важное отличие чисел с плавающей запятой от действительных чисел — это их плотность. Между любыми двумя действительными числами x и y существует другое действительное число независимо от того, насколько близко к y расположен x. Это свойство вытекает из того, что между любыми различными действи-

тельными числами x и y существует действительное число $z=(x+y)/2$. Действительные числа формируют континуум.

Числа с плавающей точкой континуума не формируют. В двухзнаковой пятиразрядной системе можно выразить ровно 179100 положительных чисел, 179100 отрицательных чисел и 0 (который можно выразить разными способами), то есть всего 358201 чисел. Из бесконечного числа действительных чисел в диапазоне от -10^{+100} до $+0{,}999\times10^{99}$ в этой системе можно выразить только 358201 число. На рис. Б.1 эти числа показаны точками. Результат вычислений может быть и другим числом, даже если он находится в области 2 или 6. Например, результат деления числа $+0{,}100\times10^{3}$ на 3 нельзя выразить *точно* в нашей системе представления. Если полученное число нельзя выразить в используемой системе представления, нужно брать ближайшее число, которое представимо в этой системе. Такой процесс называется **округлением**.

Промежутки между смежными числами, которые можно выразить в представлении с плавающей запятой, во второй и шестой областях не постоянны. Промежуток между числами $+0{,}998\times10^{99}$ и $+0{,}999\times10^{99}$ гораздо больше промежутка между числами $+0{,}998\times10^{0}$ и $+0{,}999\times10^{0}$. Однако если промежутки между числом и его соседом выразить как процентное отношение от этого числа, большой разницы в промежутках не будет. Другими словами, **относительная погрешность**, полученная при округлении, приблизительно равна и для малых, и для больших чисел.

Выводы, сделанные для системы представления с трехразрядной мантиссой и двухразрядной экспонентой, справедливы и для других систем представления чисел. При изменении числа разрядов в мантиссе или экспоненте просто сдвигаются границы второй и шестой областей и меняется число представляемых единиц в этих областях. С увеличением числа разрядов в мантиссе увеличивается плотность элементов и, следовательно, точность приближений. С увеличением количества разрядов в экспоненте размер областей 2 и 6 увеличивается за счет уменьшения областей 1, 3, 5 и 7. В табл. Б.1 показаны приблизительные границы области 6 для десятичных чисел с плавающей точкой с различным количеством разрядов в мантиссе и экспоненте.

Таблица Б.1. Приблизительные верхняя и нижняя границы чисел с плавающей точкой

Количество разрядов в мантиссе	Количество разрядов в экспоненте	Нижняя граница	Верхняя граница
3	1	10^{-12}	10^{9}
3	2	10^{-102}	10^{99}
3	3	10^{-1002}	10^{999}
3	4	10^{-10002}	10^{9999}
4	1	10^{-13}	10^{9}
4	2	10^{-103}	10^{99}
4	3	10^{-1003}	10^{999}
4	4	10^{-10003}	10^{9999}
5	1	10^{-14}	10^{9}
5	2	10^{-104}	10^{99}
5	3	10^{-1004}	10^{999}
5	4	10^{-10004}	10^{9999}
10	3	10^{-1009}	10^{999}
20	3	10^{-1019}	10^{999}

Вариант такого представления применяется в компьютерах. Основа возведения в степень — 2, 4, 8 или 16, но не 10. В этом случае мантисса состоит из цепочки двоичных, четверичных, восьмеричных и шестнадцатеричных разрядов. Если крайний левый разряд равен 0, все разряды можно сместить на один влево, а экспоненту уменьшить на 1, не меняя при этом значения числа (исключение составляет ситуация потери значимости). Мантисса с ненулевым крайним левым разрядом называется **нормализованной**.

Нормализованные числа обычно предпочитаются ненормализованным, поскольку существует только одна нормализованная форма, а ненормализованных форм может быть много. Примеры нормализованных чисел с плавающей точкой даны на рис. Б.2. для двух основ возведения в степень. В этих примерах показана 16-битная мантисса (включая знаковый бит) и 7-битная экспонента. Запятая находится слева от крайнего левого бита мантиссы и справа от экспоненты.

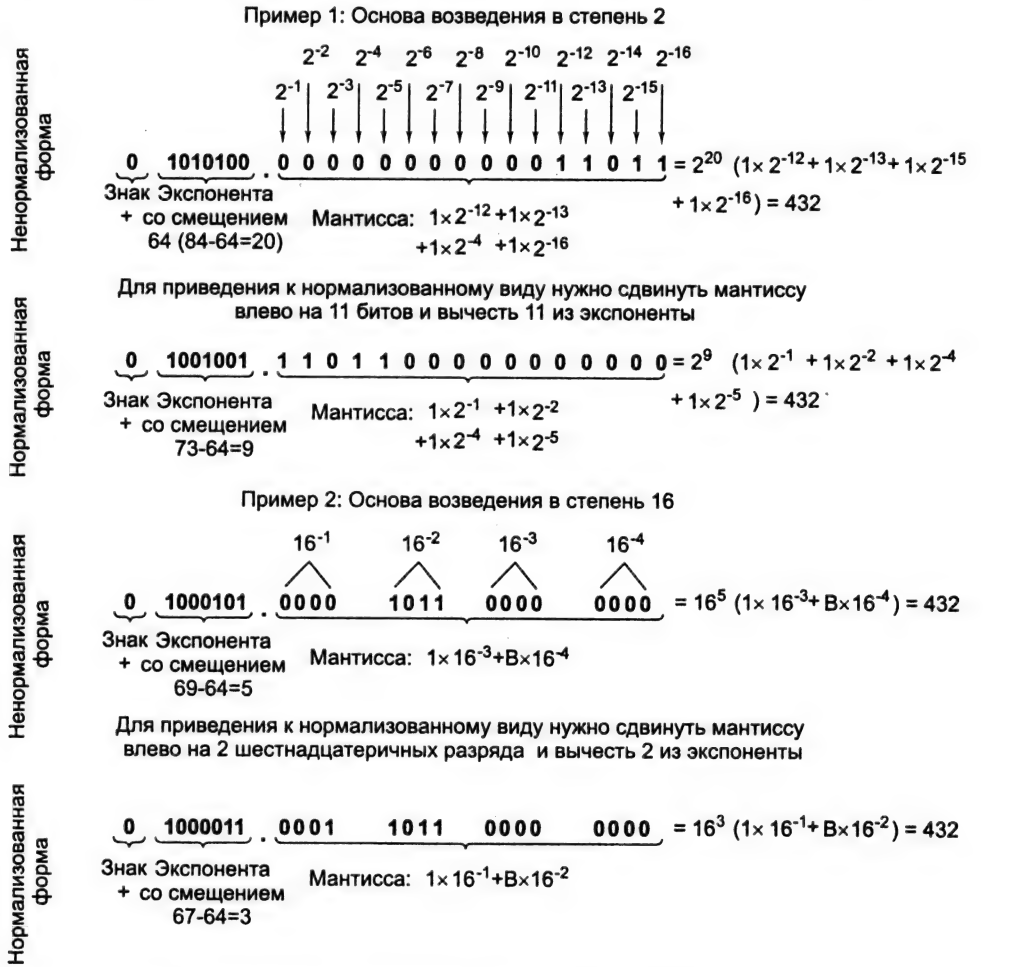


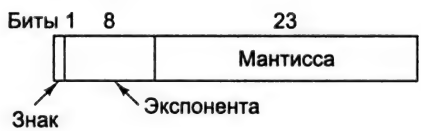
Рис. Б.2. Примеры нормализованных чисел с плавающей точкой

Стандарт IEEE 754

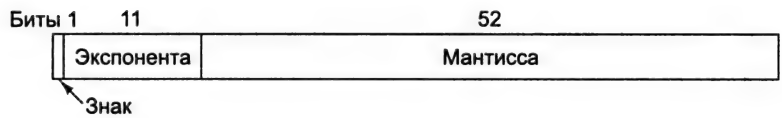
До 80-х годов каждый производитель имел свой собственный формат чисел с плавающей точкой. Все они отличались друг от друга. Более того, в некоторых из них арифметические действия выполнялись неправильно, поскольку арифметика с плавающей точкой имеет некоторые тонкости, которые не очевидны для обычного разработчика аппаратного обеспечения.

Чтобы изменить эту ситуацию, в конце 70-х годов IEEE учредил комиссию для стандартизации арифметики с плавающей точкой. Целью было не только дать возможность переносить данные с одного компьютера на другой, но и обеспечить разработчиков аппаратного обеспечения заведомо правильной моделью. В результате получился стандарт IEEE 754 (IEEE, 1985). В настоящее время большинство процессоров (в том числе Intel, SPARC и JVM) содержат команды с плавающей точкой, которые соответствуют этому стандарту. В отличие от многих стандартов, которые представляли собой неудачные компромиссы и мало кого устраивали, этот стандарт неплох, в большей степени благодаря тому, что его изначально разрабатывал один человек, профессор математики университета Беркли Вильям Каган (William Kahan). Этот стандарт будет описан ниже.

Стандарт определяет три формата: с одинарной точностью (32 бита), с удвоенной точностью (64 бита) и с повышенной точностью (80 битов). Формат с повышенной точностью предназначен для сокращения ошибок округления. Он применяется главным образом в арифметических устройствах с плавающей точкой, поэтому мы не будем о нем говорить. В форматах с одинарной и удвоенной точностью применяется основание возведения в степень 2 для мантисс и смещенная экспонента. Форматы представлены на рис. Б.3.



а



б

Рис. Б.3. Форматы для стандарта IEEE с плавающей точкой: одинарная точность (*а*); удвоенная точность (*б*)

Оба формата начинаются со знакового бита для всего числа; 0 указывает на положительное число, а 1 — на отрицательное. Затем следует смещенная экспонента. Для формата одинарной точности смещение (excess) 127, а для формата удвоенной точности смещение 1023. Минимальная (0) и максимальная (255 и 2047)

экспоненты не используются для нормализованных чисел. У них есть специальное предназначение, о котором мы поговорим ниже. В конце идут мантиссы по 23 и 52 бита соответственно.

Нормализованная мантисса начинается с двоичной запятой, за которой следует 1 бит, а затем остаток мантиссы. Следуя практике, начатой с компьютера PDP-11, компьютерщики осознали, что 1 бит перед мантиссой сохранять не нужно, поскольку можно просто предполагать, что он есть. Следовательно, стандарт определяет мантиссу следующим образом. Она состоит из неявного бита, который всегда равен 1, неявной двоичной запятой, за которыми идут 23 или 52 произвольных бита. Если все 23 или 52 бита мантиссы равны 0, то мантисса имеет значение 1,0. Если все биты мантиссы равны 1, то числовое значение мантиссы немного меньше, чем 2,0. Во избежание путаницы в английском языке для обозначения комбинации из неявного бита, неявной двоичной запятой и 23 или 52 явных битов вместо термина «мантисса» (mantissa) используется термин significand. Все нормализованные числа имеют significand s в диапазоне $1 \le s < 2$.

Числовые характеристики стандарта IEEE для чисел с плавающей точкой даны в табл. Б.2. В качестве примеров рассмотрим числа 0,5, 1 и 1,5 в нормализованном формате с одинарной точностью. Они представлены шестнадцатеричными числами 3F000000, 3F800000 и 3FC00000 соответственно.

Таблица Б.2. Характеристики чисел с плавающей точкой стандарта IEEE

Параметр	Одинарная точность	Удвоенная точность
Количество битов в знаке	1	1
Количество битов в экспоненте	8	11
Количество битов в мантиссе	23	52
Общее число битов	32	64
Смещение экспоненты	Смещение (excess) 127	Смещение (excess) 1023
Область значений экспоненты	От –126 до +127	от –1022 до +1023
Самое маленькое нормализованное число	2^{-126}	2^{-1022}
Самое большое нормализованное число	$\approx 2^{128}$	$\approx 2^{1024}$
Диапазон десятичных дробей	$\approx 10^{-38}$ до 10^{38}	$\approx 10^{-308}$ до 10^{308}
Самое маленькое ненормализованное число	$\approx 10^{-45}$	$\approx 10^{-324}$

Традиционные проблемы, связанные с числами с плавающей точкой, — что делать с переполнением, потерей значимости и неинициализированными числами. Подход, используемый в стандарте IEEE, отчасти заимствован от машины CDC 6600. Помимо нормализованных чисел в стандарте предусмотрено еще 4 типа чисел (рис. Б.4).

Проблема возникает в том случае, если абсолютное значение (модуль) результата меньше самого маленького нормализованного числа с плавающей точкой, которое можно представить в этой системе. Раньше аппаратное обеспечение действовало одним из двух способов: либо устанавливало результат на 0, либо вызывало ошибку из-за потери значимости. Ни один из этих двух способов не является удовлетворительным, поэтому в стандарт IEEE введены **ненормализованные числа**. Эти числа имеют экспоненту 0 и мантиссу, представленную следующими 23 или

52 битами. Неявный бит 1 слева от двоичной запятой превращается в 0. Ненормализованные числа можно легко отличить от нормализованных, поскольку у последних не может быть экспоненты 0.

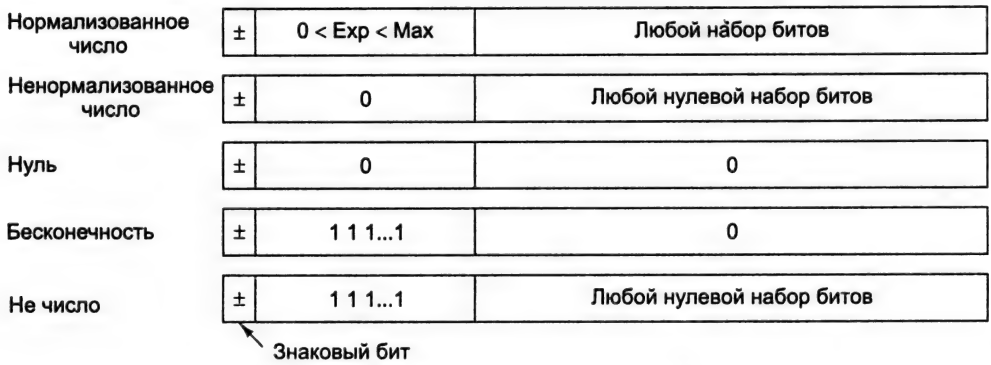


Рис. Б.4. Числовые типы стандарта IEEE

Самое маленькое нормализованное число с одинарной точностью содержит 1 в экспоненте и 0 в мантиссе и представляет $1{,}0\times2^{-126}$. Самое большое ненормализованное число содержит 0 в экспоненте и все единицы в мантиссе и представляет примерно $0{,}9999999\times2^{-127}$, то есть почти то же самое число. Следует отметить, что это число содержит только 23 бита значимости, а все нормализованные числа — 24 бита.

По мере уменьшения результата при дальнейших вычислениях экспонента попрежнему остается равной 0, а первые несколько битов мантиссы превращаются в нули, что сокращает и значение, и число значимых битов мантиссы. Самое маленькое ненулевое ненормализованное содержит 1 в крайнем правом бите, а все остальные биты равны 0. Экспонента представляет 2^{-127}, а мантисса — 2^{-23}, поэтому значение равно 2^{-150}. Такая схема предусматривает постепенное исчезновение значимых разрядов, а не перескакивает на 0, когда результат нельзя выразить в виде нормализованного числа.

В этой схеме присутствуют 2 нуля, положительный и отрицательный, определяемые по знаковому биту. Оба имеют экспоненту 0 и мантиссу 0. Здесь тоже бит слева от двоичной запятой по умолчанию 0, а не 1.

С переполнением нельзя справиться постепенно. Вместо этого существует специальное представление бесконечности: с экспонентой, содержащей все единицы, и мантиссой, равной 0. Это число можно использовать в качестве операнда. Оно подчиняется обычным математическим правилам для бесконечности. Например, бесконечность и любое число в сумме дают бесконечность. Конечное число разделить на бесконечность равно 0. Любое конечное число, разделенное на 0, стремится к бесконечности.

А что получится, если бесконечность разделить на бесконечность? Результат не определен. Для такого случая существует другой специальный формат, **NaN (Not a Number — не число)**. Его тоже можно использовать в качестве операнда.

Вопросы и задания

1. Преобразуйте следующие числа в формат стандарта IEEE с одинарной точностью. Результаты представьте в восьми шестнадцатеричных разрядах.
   а. 9
   б. 5/32
   в. −5/32
   г. 6.125
2. Преобразуйте следующие числа с плавающей точкой одинарной точности из шестнадцатеричной в десятичную систему счисления:
   а. 42E28000H
   б. 3F880000H
   в. 00800000H
   г. C7F00000H
3. Число с плавающей точкой в формате одинарной точности в IBM/370 состоит из 7-битной смещенной экспоненты (смещение 64), 24-битной мантиссы и знакового бита. Двоичная запятая находится слева от мантиссы. Основание возведения в степень — 16. Порядок полей — знаковый бит, экспонента, мантисса. Выразите число 7/64 в виде нормализованного шестнадцатеричного числа в этой системе.
4. Следующие двоичные числа с плавающей точкой состоят из знакового бита, смещенной экспоненты (смещение 64) с основанием 2 и 16-битной мантиссы. Нормализуйте их.
   а. 0 1000000 0001010100000001
   б. 0 0111111 0000001111111111
   в. 0 1000011 1000000000000000
5. Чтобы сложить два числа с плавающей точкой, нужно уровнять экспоненты (сдвинув мантиссу). Затем можно сложить мантиссы и нормализовать результат, если в этом есть необходимость. Сложите числа одинарной точности 3EE00000H и 3D800000H и выразите нормализованный результат в шестнадцатеричной системе счисления.
6. Компьютерная компания решила выпустить машину с 16-битными числами с плавающей точкой. В модели 0.001 формат состоит из знакового бита, 7-битной смещенной экспоненты (смещение 64) и 8-битной мантиссы. В модели 0.002 формат состоит из знакового бита, 5-битной смещенной экспоненты (смещение 16) и 10-битной мантиссы. В обеих моделях основание возведения в степень равно 2. Каково самое маленькое и самое большое положительное нормализованное число в этих моделях? Сколько десятичных разрядов точности содержится в каждой модели? А вы купили бы какую-нибудь из этих двух моделей?

7. Существует одна ситуация, при которой операция над двумя числами с плавающей точкой может вызвать сильное сокращение количества значимых битов в результате. Что это за ситуация?
8. Некоторые микросхемы с плавающей точкой имеют встроенную команду квадратного корня. Возможно применение итерационного алгоритма (например, метода Ньютона—Рафсона). Итерационные алгоритмы дают последовательные приближения решения. Как можно быстро получить приближенный квадратный корень от числа с плавающей точкой?
9. Напишите процедуру сложения двух чисел одинарной точности с плавающей точкой. Каждое число представлено 32-элементным логическим массивом.
10. Напишите процедуру сложения двух чисел с плавающей точкой одинарной точности, в которых для экспоненты используется основание системы счисления 16, а для мантиссы — основание системы счисления 2 и которые не содержат неявного бита 1 слева от двоичной точки. В нормализованном числе крайние левые 4 бита мантиссы могут быть 0001, 0010, ..., 1111, но не 0000. Число нормализуется путем сдвига мантиссы влево на 4 бита и прибавления 1 к экспоненте.

Алфавитный указатель

A

ACL, список контроля доступа, 502
APIC, 196
ASCII-код, 130
ATM, асинхронный режим передачи, 630
attraction memory, 619

B

BGA, Ball Grid Array, 205
BIOS, базовая система ввода-вывода, 91
BIPUSH, команда IJVM, 248, 265
Burroughs B5000, 37

C

CC-NUMA, 607
CD-ROM XA, 103
CDC 6600, 68, 308, 578
CDC-6600, 36
Celeron, 47
CISC, компьютер с полным набором команд, 63
COLOSSUS, 31
COMA, 586, 619
Control Data Corporation, 36
COW, кластер рабочих станций, 44, 586, 626
CPP, регистр, 247, 258
Cray T3E, 623
Cray-1, 588
CRC, циклический избыточный код, 218
CYMK, 124

D

DAS, 627
DASH, 611
Digital Equipment Corporation, 35, 61
DIMM, 84
DIP, двурядный корпус, 150
DLL, динамически подключаемая библиотека, 549
DMA, прямой доступ к памяти, 109
dpi, 122
DSM
  аппаратная, 608
DSM, распределенная совместно используемая память, 562, 636
DUP, команда IJVM, 248, 265
DVD-диск, 105

E

E-регистр, 623
eagle, плата, 625
EDVAC, 33
EIDE-диски, 92
EISA, расширенная ISA, 110
ENIAC, 33
ENIGMA, 32
EPIC, 425
Ethernet, 628
  с использованием коммутаторов, 630
excess, система представления чисел, 672
exe-файл, 540

F

FAT
  см. таблица размещения файлов, 499
FIFO, алгоритм, 447
FMS, FORTRAN Monitor System, 26
FORTRAN, 26

G

GDT, глобальная таблица дескрипторов, 455
Gigaplahe-XB, 604
GigaRing, 624
Globe, 642
GOTO, команда IJVM, 248, 268
goto, оператор микроассемблера, 257

H

H регистр, 233

I

IA-32, 316, 343
IA-64, 425
IADD, команда IJVM, 248, 260, 264
IAND, команда IJVM, 248, 265
IAS, 33
IBM PC, происхождение, 39
IBM PS/2, 206
IBM, корпорация, 35
IBM-1401, 36
IBM-360, 38
IBM-701, 35
IBM-704, 35
IBM-709, 26
IBM-7094, 36, 38
IBM-801, 62
IDE-диск, 91
IF_ICMPEQ, команда IJVM, 248, 270
IFEQ, команда IJVM, 248, 270
IFLT, команда IJVM, 248, 270
IINC, команда IJVM, 249, 268
IJVM, 230, 244
  Java, 252
  набор команд, 248
  реализация Mic-2, 280
ILC, счетчик адреса команд, 532
ILLIAC, 33
ILLIAC IV, 585
ILLIAC IV, 70
ILOAD, команда IJVM, 248, 265
Intel 8255A, 219
Intel 8259A, 192
Intel Pentium, 46
Intel, корпорация, 45
Intel-4004, 45
Intel-8008, 45
Intel-80286, 46
Intel-80386, 46
Intel-80486, 46
Intel-8080, 45
Intel-8086, 45
Intel-8088, 45
INVOKEVIRTUAL, команда IJVM, 249, 271
IOR, команда IJVM, 248, 265
IQ-Link, плата, 616
IRETURN, команда IJVM, 251, 271
ISA, стандартная промышленная архитектура, 110
ISDN, 128
ISTORE, команда IJVM, 248
ISUB, команда IJVM, 248, 265
IU, процессор целочисленной арифметики, 50

J

JIT-компилятор, 51
JOHNIAC, 33
JVM, виртуальная машина Java, 51

K

kestrel, плата, 625

L

Latin-1, 132
LBA, 92
LDC_W, команда IJVM, 248, 268
LDT, локальная таблица дескрипторов, 455
Linda, 638
lpi, 124
LRU, алгоритм, 299, 446
LV, регистр, 245, 247, 250, 258

M

MAL, микроассемблер, 255
MAR, регистр адреса ячейки памяти, 236

MBR, буферный регистр памяти, 236, 258, 266, 277
MDR, информационный регистр памяти, 236
Merced, 425
MESI, протокол, 601
MFT, главная файловая таблица, 502
Mic-1, 240
Mic-2, 280
  микропрограмма, 280
Mic-3, 285
Mic-4, 290
microJava II 701, цифровой логический уровень, 204
MicroJava, введение, 52
Microsoft, 40
MIMD, 585
MIPS
  микросхема, 62
  число миллионов команд в секунду, 64
MIR, регистр микрокоманд, 240
MISD, 585
MMU, контроллер управления памятью, 442
MMX, 47
Motif, 483
Motorola 68000, 62
MPC, микропрограммный счетчик, 240
MPI, интерфейс с передачей сообщений, 634
MPP, процессор с массовым параллелизмом, 586, 622
MULTICS, 454, 521
Myrinet, 631

N

NaN, 682
NC-NUMA, 607
NIC, сетевой адаптер, 625
NOP, команда IJVM, 249, 264
NORMA, 586
NOW, сеть рабочих станций, 44, 586, 626
NTFS
  см. файловая система Windows NT, 499
NUMA, 586, 607
NUMA-Q, 615

O

OC-12, 630
omega, сеть, 605
Omnibus, 35
OPC, регистр, 259
Option Blue, 625
Option Red, 625
Option White, 625
Orca, 640

P

PC
  регистр, 251, 258
  счетчик команд, 247
PCI, 110
PDP-1, 35
PDP-8, 35
Pentium II
  блок
    возврата, 318
    вызова/декодирования, 315
    отправки/выполнения, 317
  виртуальный режим 8086, 343
  компоновка, 194
  конвейерный режим, 198
  реальный режим, 343
  регистры, 343
  управление режимом электропитания, 197
  цоколевка, 195
Pentium II
  введение, 47
  цифровой логический уровень, 194
picoJava I, 51
picoJava II
  конвейер, 324
  микроархитектура, 323
picoJava II
  цифровой логический уровень, 204
picoJava II, введение, 51
PIO, параллельный ввод-вывод, 219
poison bit, 314
POP, команда IJVM, 248, 259, 265
POSIX, 480
POSIX, подсистема Windows NT, 487
PSW, слово состояния программы, 341
pthreads, 506
PVM, виртуальная машина параллельного действия, 633

Q

quard board, плата, 615

R

RAID, 94
RAW-взаимозависимость, 288
replicated worker, алгоритм, 582, 639
RISC, компьютер с сокращенным набором команд, 40, 63
  принципы разработки, 64
ROB, буфер переупорядочивания команд, 314
RS-232-C, терминал, 117

S

SCI, масштабируемый когерентный интерфейс, 615
SCSI, 92
SEC, Single Edge Cartridge, 194
Sequent NUMA-Q, 615
SIB, масштаб, индекс, база, 359, 376
SID, идентификатор безопасности, 502
SIMD, 585, 587
SIMM, 84
Single Edge Cartridge, 194
SISD, 584
SLED, 94
SMP, симметричный мультипроцессор, 593
SO-DIMM, 85
SP, регистр, 231, 245, 251, 258
SPARC, 49
SPMD, 581
Sun Enterprise 10000, 604
Sun Microsystems, 48
SWAP, команда IJVM, 248, 265, 287

T

TAT-12/13, 42
TLB, буфер быстрого преобразования адреса, 460
TOS, регистр, 259
TSB, буфер хранения преобразований, 462
TX-0, 35

U

u-конвейер, 68
UART, универсальный асинхронный приемопередатчик, 118, 219
UDB II, UltraSPARC II Data Buffer II, 202
UltraSPARC I, 50
UltraSPARC II, 50
  конвейер, 321
UltraSPARC II
  цифровой логический уровень, 200
UMA, 586
UNICODE, 132
UNIX, 480
  Berkeley, 480
  Solaris, 481
  System V, 480
UPA, высокоскоростной пакетный коммутатор, 201
USART, 219

V

v-конвейер, 68
VAX, 61, 63
VIS, 50
VTOC, оглавление диска, 104

W

WAR-взаимозависимость, 309
WAW-взаимозависимость, 309
WEIZAC, 33
Whirlwind, 34
WIDE, команда IJVM, 249, 267
Win32 API, 488
Win32, подсистема Windows NT, 487
Windows, 483
Windows 95, 483
Windows 98, 483
Windows NT, 484
wormhole routing, «червоточина», 571

X

X Windows, 483
Xeon, 47

Z

Zilog Z8000, 62

А

автомат с конечным числом состояний, 278
прогнозирование переходов, 304
автономная
рабочая станция, 627
автономная информация, 469
аддитивная инверсия, 383
адрес памяти, 74
адресация, 353
JVM, 377
Pentium II, 375
UltraSPARC II, 377
индексная, 367
команды перехода, 372
косвенная регистровая, 366
непосредственная, 365
относительная индексная, 369
прямая, 366
регистровая, 366
способы адресации, 365
стековая, 369
адресное пространство, 439
Айкен, Говард, 32
аккумулятор, 34, 59, 365
активное ожидание, 388
активный матричный индикатор, 115
алгебра релейных схем, 142
алгоритм, 24
АЛУ, арифметико-логическое устройство, 22, 57, 159, 231
Амдала закон, 575
аналитическая машина, 30
аппаратное обеспечение, 24
арбитр шины, 109
арбитраж шины, 188
арифметико-логическое устройство, 22, 57, 159, 231
архитектура, 24, 60
загрузки/хранения, 348
компьютерная, 24
асинхронный режим передачи, 630
ассемблер, 23, 518
таблица символьных имен, 537
ассоциативная память, 454, 538
Атанасов, Джон, 32
атрибутивный байт, 116
аудио-видеодиск, 89

Б

база, 140
базовая система ввода-вывода, 91
базовый элемент, 311
байт, 75, 338
барьер, 584, 635
Бехтольсхайм, Энди, 48
библиотека импорта, 550
библиотека коллективного доступа, 551
бинарные операции, 380
бисекционная пропускная способность, 565
бит, 73, 667
четности, 78
бит присутствия, 443
битовое отображение, 351
блок
выборки команд, 277
декодирования, 290
формирования очереди, 290
блок двойной косвенной адресации, 498
блок косвенной адресации, 498
блок тройной косвенной адресации, 498
блокировка начала очереди, 570
блокируемая сеть, 607
бод, 128
большие компьютеры, 44
булева алгебра, 142
Буль, Джордж, 142
буфер
выборки с упреждением, 66
переупорядочивания команд, 314
буфер быстрого преобразования адреса, 460
буфер хранения преобразований, 461
буферизация
на входе, 570
на выходе, 570
общая, 570
буферный
регистр памяти, 236, 258, 266, 277
элемент
без инверсии, 171
с инверсией, 171
Бэббидж, Чарльз, 30

В

ввод-вывод с распределением
памяти, 221
вектор, 588
вектор прерываний, 414
вектор прерывания, 193
векторный
процессор, 70, 588
регистр, 71
вентиль, 21, 139
взаимное исключение, 583
взаимоблокировка, 571
видео-ОЗУ, 116
видеопамять, 115
винчестер, 88
виртуальная
машина
параллельного действия, 633
топология, 635
виртуальная машина, 19
Java, 51
виртуальная память, 439
Pentium II, 455
UltraSPARC II, 460
виртуальное адресное
пространство, 440
виртуальный регистр, 244
внешний символ, 544
внешняя ссылка, 543
Возняк, Стив, 39
восьмеричная система счисления, 667
впадина, 98
временная локализация, 295
время
ожидания сектора, 88
такта, 161
время принятия решения, 545
входной язык, 517
выбора маршрута алгоритм, 571
выборка-декодирование-
исполнение, 59
выборка-исполнение, цикл, 231
выделенная страница, 491
вызов страниц по требованию, 445
вызов супервизора, 27
выравнивание по правому биту, 380
высокоскоростной пакетный
коммутатор, 201
выходной язык, 517
вычислительный центр, 39

Г

гарвардская архитектура, 84
гиперкуб, 567
главная библиотека, 551
главная файловая таблица, 502
глобальная таблица дескрипторов, 455
градация полутонов, 124
графический интерфейс
пользователя, 483
графический терминал, 116

Д

двоичная система счисления, 667
двоично-десятичный код, 73
двоичный поиск, 538
двойной тор, 567
двурядный корпус, 150
двухпроходной транслятор, 532
двухточечная передача сообщений, 583
Де Моргана законы, 146
декодер, 153
декодирование адреса частичное, 223
демультиплексор, 153
дерево, 567
дескриптор защиты, 502
дескриптор файла, 493
Джобс, Стив, 39
Джой, Билл, 48
диаметр сети межсоединений, 565
дибитная фазовая кодировка, 127
динамически подключаемая
библиотека, 549
динамическое связывание, 547
директива ассемблера, 525
директория, 469
диск, 86
дискета, 90
диспетчер безопасности,
Windows NT, 487
диспетчер ввода-вывода,
Windows NT, 486
диспетчер кэш-памяти, Windows NT, 486
диспетчер объектов, Windows NT, 486

диспетчер процессов и потоков, Windows NT, 487
длина пути, 272
  сокращение, 274
дополнение
  до двух, 672
  до единицы, 672
дорожка, 87
драйвер устройств, Windows NT, 486
драйвер шины, 181
дрибблинг, 324
дробь, 676
дуплексный, 128

Ж

желтая книга, 100
жидкокристаллический дисплей, 113

З

задающее устройство шины, 180
задержка вентиля, 151
закон Мура, 41
замкнутость, 666
занятие цикла памяти, 110
заполнение по записи, политика, 601
запоминающее устройство, 73
запуск
  уровнем сигнала, 166
  фронтом сигнала, 166
зарезервированная страница, 491
захват цикла, 390
защелка
  SR-защелка, 163
  синхронная
    D-защелка, 165
    SR-защелка, 164
звезда, 567
Зеленая книга, 101
знаковое расширение, 237
«зуб вампира», 629
Зус, Конрад, 31

И

идентификатор, 488
идентификатор безопасности, 502
иерархическая структура памяти, 85
инвертирующие выходы, 141
инвертор, 141
индекс файла, 466
индексация цветов, 117
индексный дескриптор, 497
интегральная схема, 149
  применение в компьютерных системах, 37
интервал Хэмминга, 77
интерпретатор, 19, 60
интерпретация, 19
интерфейс
  с передачей сообщений, 634
интерфейс графических устройств, Windows NT, 487
инфиксная запись, 369
информационное ядро, 616
информационный регистр памяти, 236
ИС, интегральная схема, 149
исполнение с изменением последовательности, 310
исполняемая двоичная программа, 517
исполняемый двоичный код, 540
исполняющая система, Windows NT, 486

К

канал, UNIX, 504
канальная карта, 630
квитирование полное, 187
клавиатура, 112
кластер рабочих станций, 586, 626
клон, 40
ключ, 466
код
  операции, 231
  Рида—Соломона, 87
  с исправлением ошибок, 77
  символа, 129
  смены алфавита, 359
  условия, 341
  Хэмминга, 79
кодированное слово, 77
кодовая страница, 132
коллектор, 140
кольцевой буфер, 472
кольцо, 567
команды
  ввода-вывода, 386
  перемещения, 379

команды (*продолжение*)
сравнения, 384
условного перехода, 383
комбинационная схема, 151
коммуникатор, 634
коммутация
без буферизации пакетов, 570
каналов, 569
с промежуточным хранением, 569
компакт-диск, 98
CD-R, 102
CD-ReWritable, 105
CD-RW, 105
дорожка, 103
многосессионный, 104
сектор, 100
фрейм, 100
компаратор, 154
компилятор, 24, 518
компоновщик, 539
компьютер
параллельного действия, 556
с полным набором команд, 63
с сокращенным набором команд, 40, 63
компьютерная организация, 24
конвейер, 66, 581
Pentium, 68
конвейерная модель (Mic-3), 285
конечной точности числа, 665
конечный автомат, 278
константа перемещения, 543
контекст, 461
контроллер, 108
диска, 91
последовательности, 240
контроллер управления памятью, 442
контрольная задача, 520
координатный коммутатор, 603
корневой каталог, 495
кортеж, 638
косвенная (слабая) связь системы, 558
Косла, Винолд, 48
коэффициент
разветвления, 565
коэффициент совпадения, 83
Красная книга, 98
Крей, Сеймур, 36
критическая секция, 509
куб, 567
кугуар, 626
куча, 349
кэш-память, 46, 82, 295
ассоциативная n-входовая, 299
второго уровня, 295
заполнение по записи, 300
обратная запись, 300
прямого отображения, 296
разделенная, 84, 295
с отслеживанием, 599
сквозная запись, 300
смежная, 84
кэш блоков, 482

Л

линейный адрес, 456
литерал, 534
Ловлейс, Ада, 30
ловушка, 412
логическая запись, 465
ложное совместное использование, 637
локальная таблица дескрипторов, 455

М

магнитный диск, 87
МакНили, Скот, 48
макроархитектура, 245
макровызов, 527
макроопределение, 527
макрорасширение, 527
макрос, 527
фактические параметры, 529
формальные параметры, 529
мантисса, 676
маркер доступа, 501
маршрутизация
адаптивная, 572
от источника, 571
пространственная, 572
статическая, 572
маска, 380
массивно-параллельный процессор, 70, 587
масштаб, индекс, база, 359, 376
масштабируемый когерентный интерфейс, 615

материнская плата, 108
машинный язык, 18
межсекторный интервал, 87
металл-оксид-полупроводник, 142
микроассемблер, 255
микрокоманда, 62
микрооперация, 291
микропрограмма, 22
микропрограммирование, 22
микропрограммный счетчик, 240
микросхема, 149
  процессора, 177
микроядро, Windows NT, 486
Мирвольд, Натан, 42
многопоточная обработка, 578
многоступенчатые сети, 605
многоуровневая компьютерная организация, 18
модем, 118
модуль управления виртуальной памятью, Windows NT, 486
модуляция, 127
  амплитудная, 127
  фазовая, 127
  частотная, 127
монитор, 112
монтажное ИЛИ, 181
Моушли, Джон, 32
мультивещание, 583
мультикомпьютер, 72, 560, 586, 621
  расширяемый, 561
мультиплексор, 151
мультипрограммирование, 38
мультипроцессор, 71, 559, 586
  на основе каталога, 609
Мур, Гордон, 41, 45
мышь, 119
мьютекс, 506

Н

набор констант, IJVM, 247
настройка, 520
неавтономная информация, 469
неблокируемая сеть, 603
негативная логика, 149
ненормализованное число, 681
непосредственная (тесная) связь системы, 558
непосредственный операнд, 365
непротиворечивость кэшей, 598
несущий сигнал, 127
нить, 507, 508
Нойс, Роберт, 37, 45
нормализованное число, 679

О

область процедур, IJVM, 247
оболочка, 483, 623
обработка полутонов, 123
обработчик системных прерываний, 413
обратная польская запись, 369
обратно совместимый, 335
объектная программа, 517
объектный модуль, 540, 543
оверлей, 438
оглавление диска, 104
одноразрядные секции, 161
окно, 117
округление, 678
Ольсен, Кеннет, 35
операционная система, 26, 437
операция (Orca), 640
опережающая ссылка, 532
оптимальной подгонки алгоритм, 453
оптический диск, 98
Оранжевая книга, 103
основание системы счисления, 667
открытый коллектор, 181
относительная погрешность, 678
отсрочка ветвления, 302
очередь сообщений, 505
ошибка
  из-за потери значимости, 677
  переполнения, 677
ошибка из-за отсутствия страницы, 445

П

пакет, 564
  задач, 639
пакетный режим, 28
память, 73, 163
  EDO, 175
  FPM, 174

память (*продолжение*)
динамическое ОЗУ, 174
обновление, 174
синхронное, 175
ОЗУ, оперативное запоминающее устройство, 174
ПЗУ, постоянное запоминающее устройство, 175
программируемое, 175
стираемое программируемое, 175
электронно-перепрограммируемое, 176
статическое ОЗУ, 174
флэш-память, 176
парадигма, 581
параллелизм на уровне
блоков, 580
команд, 65
крупных структурных единиц, 558
мелких структурных единиц, 558
процессоров, 65
параллельный ввод-вывод, 219
Паскаль, Блез, 29
пассивный матричный индикатор, 115
ПДП, прямой доступ к памяти, 389
передача сообщений
буферная, 632
неблокируемая, 632
синхронная, 632
перекос шины, 183
переменная условия, 507
перераспределения памяти проблема, 543
переход, конечный автомат, 278
период ожидания, 184
печатающее устройство, 123
пиксел, 116
планирование (в мультикомпьютере), 628
площадка, 98
подмена регистров, 311
подпрограмма, 385
подсистемы окружения, Windows NT, 487
подчиненное устройство шины, 180
позитивная логика, 149
позиционно-независимая программа, 547
поиск, 88
поклеточная разбивка, 452
полное межсоединение, 567
полный вентиль, 145
полубайт, 391
полудуплексный, 128
пользовательский режим, 338
порождающий процесс, 504
порожденный процесс, 504
последовательного опроса система, 188
постфиксная запись, 369
поток, 473, 482, 581
UNIX, 505
поток управления, 404
потребитель (процесс), 472
почтовый ящик, 509
преамбула, 87
предикация, 426
представление с плавающей точкой, 676
прерывание, 109, 413
неточное, 309
точное, 309
префикс, 394
префиксный байт, 267, 359
привилегированный пользователь, 497
привилегированный режим, 338
приемник шины, 181
приемопередатчик шины, 181
принтер, 121
лазерный, 122
матричный, 121
с восковыми чернилами, 126
с твердыми чернилами, 125
струйный, 122
принцип локальности, 82
пробуксовка, 447
программа, 18
обработки прерываний, 413
обработки прерывания, 109
чистки памяти, 349
программируемая логическая матрица, 155
программируемый ввод-вывод, 387
программное обеспечение, 24
прозрачность, 415
производитель (процесс), 472

пролог процедуры, 408
промах TLB, 461
промах кэш-памяти, 298
пропускная способность
процессора, 67
простаивание, 288
конвейера, 302
пространственная локализация, 295
пространство кортежей, 638
протокол
когерентности кэширования, 598
стратегия, 601
однократной записи, 602
с обратной записью, 601
проход ассемблера, 532
процедура, 385, 405
рекурсивная, 405
процессор
с массовым параллелизмом, 586, 622
процессор целочисленной арифметики, 50
прямой доступ к памяти, 109, 389
псевдокоманда, 524
путь, 495
абсолютный, 495
относительный, 495
пучок, 425

P

рабочее множество, 445
разгрузка оперативного запоминающего устройства, 26
«разделяй и властвуй», алгоритм, 582
размер страницы, 448
размерность сети межсоединений, 566
разметка, 94
разностная машина, 30
распределенная совместно используемая память, 562, 636
расслоенная память, 607
растровая развертка, 112
расфазировка данных, 569
расширение
кода операции, 356
по знаку, 237
расширенная ISA, 110
расширяемая система, 576
реальная взаимозависимость, 288
регистр, 21, 168
адреса ячейки памяти, 236
команд, 56
микрокоманд, 240
уровень архитектуры команд, 340
регистровые окна, 346
редактор связей, 539
рекурсия, 385
решетка, 567

C

самоизменяющаяся программа, 367
СБИС, сверхбольшая интегральная схема, 39
сборщик мусора, программа чистки памяти, 349
сбросить сигнал, 173
свертывание команд, 325
светодиод, 120
свободная страница, 491
связывание
неявное, 550
явное, 551
связывающий загрузчик, 539
связь, 495
сдвиговый регистр динамики переходов, 305
сегмент, 450
сегмент связи, 547
сектор, 87
семафор, 476
сессия компакт-диска, 104
сетевой
адаптер, 625
концентратор, 629
сетка, 567
сеть
межсоединений, 564
рабочих станций, 586, 626
сеть рабочих станций, 44
сигнал управления, 236
символьный терминал, 115
симплексный, 128
синхронизация шины, 183

система
с распределенной памятью, мультикомпьютер, 560
с совместно используемой памятью, 559
системные службы, Windows NT, 487
системный
вызов, 27
программист, 23
системный вызов, 437
системный интерфейс, 487
системы с разделением времени, 28
сквозное кэширование, 599
скрученный нематик, 114
слово, 75
состояния программы, 341
слово состояния программы
Pentium II, 458
события, 510
согласованность
модели, 593
по последовательности, 593
процессорная, 595
свободная, 596
слабая, 596
строгая, 593
соединение, 603
сокет, 481
сопрограмма, 411
состояние
компьютера, 231
конечный автомат, 278
состояние гонок, 476
спекулятивная загрузка, 429
спекулятивное выполнение, 313
список контроля доступа, 502
список свободной памяти, 467
стадия конвейера, 66
стандартная ошибка, 495
стандартная промышленная архитектура, 110
стандартный ввод, 495
стандартный вывод, 495
стек
IJVM, 245
операндов, 246, 253
IJVM, 247
степень, 565
детализации, 558
Стибитс, Джон, 32
страница, 440
страничная организация памяти, 440
страничный
сканер, 608
страничный кадр, 441
стробировать, 165
строка кэш-памяти, 83, 296
сублимация, 126
суммарная пропускная способность, 574
сумматор
полный сумматор, 158
полусумматор, 158
с выбором переноса, 159
со сквозным переносом, 158
суперкомпьютер, 36, 44
суперскалярная архитектура, 68
схема распределения памяти, 440
схема сдвига, 157
счетчик
команд, 56, 247
обращений, 308
счетчик адреса команд, 532

Т

таблица
локальной памяти, 616
таблица истинности, 142
таблица размещения файлов, 499
таблица символов, 532
таблица страниц, 440, 457
тактовый генератор, 161
тасование полное, 606
текущий каталог, 495
терминал, 111
типы данных
нечисловые, 351
числовые, 350
толстое дерево, 567
топология, 565
точка входа, 544
тракт данных, 22, 57
IJVM, 231
Mic-3, 286
транзакция шины, 198
транзистор, 35
биполярный, 142
МОП, металл-оксид-полупроводник, 142

транслирующая таблица, 462
транслятор, 517
трансляция, 19
триггер, 165
ТТЛ, транзисторно-транзисторная
логика, 142

У

удачное обращение в кэш-память, 298
Уилкс, Морис, 61
указатель, 366
фрейма, 344
указатель кода, 132
унарные операции, 381
унаследованная шина, 194
универсальный
асинхронный приемопередатчик, 219
синхронно-асинхронный
приемопередатчик, 219
универсальный асинхронный
приемопередатчик, 118
управляющая память, 240
упреждающая выборка, 578
уровень, 20
аппаратных абстракций, 485
архитектуры
команд, 334
системы команд, 23
микроархитектурный, 22, 230
разработка, 271
операционной системы, 23, 437
физических устройств, 21, 140
цифровой логический, 21
языка ассемблера, 517
условное выполнение, 427
установить сигнал, 173
устаревшие данные, 598
устройство с тремя состояниями, 171

Ф

файл, 464
непосредственный, 503
файловая система Windows NT, 499
файловая система, Windows NT, 486
физическое адресное пространство, 440
фильтр, 495
флаговый регистр, 341
Флинна классификация, 584
фон Нейман, Джон, 33
фон-неймановская вычислительная
машина, 34, 57
фрагментация
внешняя, 452
внутренняя, 448
фрейм локальных переменных, 245
IJVM, 247

Х

хаб, 629
хаб центральный, 216
Ханойская башня, 405, 417
хэш-кодирование, 538

Ц

цветовая
палитра, 117
цветовая шкала, 124
целевая библиотека, 551
центральный процессор, 56
цикл
тракта данных, 58
шины, 183
цилиндр, 88
цоколевка, 177

Ч

частотная манипуляция, 127
червоточина, 571
чернила на основе
красителя, 125
пигмента, 125
число
со знаком, 670
число с удвоенной точностью, 350

Ш

шаблон, 639
шестнадцатеричная система
счисления, 667
шина, 35, 56, 109, 179
EISA, 207
IBM PC, 206

шина (*продолжение*)
ISA, 207
PCI, 208
задающее устройство, 210
подчиненное устройство, 210
сигналы, 211
транзакция, 210
Sbus, 201
USB, универсальная последовательная шина, 216
асинхронная, 183, 186
мультиплексная, 183
протокол, 180
синхронная, 183
системная, 179
ширина шины, 182
широковещание, 583

Э

эквивалентные схемы, 145
Экерт, Дж. Преспер, 33
экспонента, 676
электронно-лучевая трубка, 112
эмиттер, 140
эпилог процедуры, 408
ЭСЛ, эмиттерно-связанная логика, 142
Эстридж, Филип, 39
эффективный цикл, 41

Я

язык ассемблера, 518
язык высокого уровня, 24
ячейка памяти, 74

Эндрю Таненбаум

Архитектура компьютера
4-е издание

Под научной редакцией проф. А. В. Гордеева

Главный редактор	*Е. Строганова*
Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Васильев*
Переводчик	*И. Ткачева*
Литературный редактор	*Е. Ваулина*
Художник	*Н. Биржаков*
Иллюстрации	*В. Шендерова*
Корректоры	*Т. Брылёва, И. Тимофеева*
Верстка	*Р. Гришанов*

Лицензия ИД № 05784 от 07.09.01.
Подписано в печать 06.08.04. Формат 70×100/16. Усл. п. л. 56,76. Доп. тираж 3000 экз. Заказ № 140.
ООО «Питер Принт». 194044, Санкт-Петербург, пр. Б. Сампсониевский, д. 29а.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 953005 — литература учебная.
Отпечатано с готовых диапозитивов в ФГУП «Печатный двор» Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

КЛУБ ПРОФЕССИОНАЛ

В 1997 году по инициативе генерального директора **Издательского дома «Питер»** Валерия Степанова и при поддержке деловых кругов города в Санкт-Петербурге был основан **«Книжный клуб Профессионал»**. Он собрал под флагом клуба профессионалов своего дела, которых объединяет постоянная тяга к знаниям и любовь к книгам. Членами клуба являются лучшие студенты и известные практики из разных сфер деятельности, которые хотят стать или уже стали профессионалами в той или иной области.

Как и все развивающиеся проекты, с течением времени книжный клуб вырос в **«Клуб Профессионал»**. Идею клуба сегодня формируют три основные «клубные» функции:

- неформальное общение и совместный досуг интересных людей;
- участие в подготовке специалистов высокого класса (семинары, пакеты книг по специальной литературе);
- формирование и высказывание мнений современного профессионала (при встречах и на страницах журнала).

КАК ВСТУПИТЬ В КЛУБ?

Для вступления в **«Клуб Профессионал»** вам необходимо:

- ознакомиться с правилами вступления в **«Клуб Профессионал»** на страницах журнала или на сайте **www.piter.com**;
- выразить свое желание вступить в **«Клуб Профессионал»** по электронной почте **postbook@piter.com** или по тел. **(812) 103-73-74**;
- заказать книги на сумму не менее 500 рублей в течение любого времени или приобрести комплект **«Библиотека профессионала»**.

«БИБЛИОТЕКА ПРОФЕССИОНАЛА»

Мы предлагаем вам получить все необходимые знания, подписавшись на **«Библиотеку профессионала»**. Она для тех, кто экономит не только время, но и деньги. Покупая комплект – книжную полку **«Библиотека профессионала»**, вы получаете:

- **скидку 15%** от розничной цены издания, без учета почтовых расходов;
- при покупке двух или более комплектов – дополнительную **скидку 3%**;
- членство в **«Клубе Профессионал»**;
- подарок – журнал **«Клуб Профессионал»**.

Закажите бесплатный журнал «Клуб Профессионал».

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ПИТЕР

Нет времени

ходить по магазинам?

наберите:

www.piter.com

Здесь вы найдете:

Все книги издательства сразу
Новые книги — в момент выхода из типографии
Информацию о книге — отзывы, рецензии, отрывки
Старые книги — в библиотеке и на CD

И наконец, вы нигде не купите наши книги дешевле!

КНИГА-ПОЧТОЙ



ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- по телефону: **(812) 103-73-74;**
- по электронному адресу: **postbook@piter.com;**
- на нашем сервере: **www.piter.com;**
- по почте: **197198, Санкт-Петербург, а/я 619, ЗАО «Питер Пост».**

ВЫ МОЖЕТЕ ВЫБРАТЬ ОДИН ИЗ ДВУХ СПОСОБОВ ДОСТАВКИ И ОПЛАТЫ ИЗДАНИЙ:

- Наложенным платежом с оплатой заказа при получении посылки на ближайшем почтовом отделении. Цены на издания приведены ориентировочно и включают в себя стоимость пересылки по почте (**но без учета авиатарифа**). Книги будут высланы нашей службой **«Книга-почтой»** в течение двух недель после получения заказа или выхода книги из печати.
- Оплата наличными при курьерской доставке (**для жителей Москвы и Санкт-Петербурга**). Курьер доставит заказ по указанному адресу в удобное для вас время в течение трех дней.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, код, количество заказываемых экземпляров.

Вы можете заказать бесплатный журнал «Клуб Профессионал»

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

СПЕЦИАЛИСТАМ КНИЖНОГО БИЗНЕСА!

ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»

предлагают эксклюзивный ассортимент компьютерной, медицинской, психологической, экономической и популярной литературы

РОССИЯ

Москва м. «Калужская», ул. Бутлерова, д. 17б, офис 207, 240; тел./факс (095) 777-54-67; e-mail: sales@piter.msk.ru

Санкт-Петербург м. «Выборгская», Б. Сампсониевский пр., д. 29а; тел. (812) 103-73-73, факс (812) 103-73-83; e-mail: sales@piter.com

Воронеж ул. 25 января, д. 4; тел. (0732) 27-18-86; e-mail: piter-vrn@vmail.ru; piterv@comch.ru

Екатеринбург ул. 8 Марта, д. 267б; тел./факс (343) 225-39-94, 225-40-20; e-mail: piter-ural@r66.ru

Нижний Новгород ул. Премудрова, д. 31а; тел. (8312) 58-50-15, 58-50-25; e-mail: piter@infonet.nnov.ru

Новосибирск ул. Немировича-Данченко, д. 104, офис 502; тел/факс (3832) 54-13-09, 47-92-93, 11-27-18, 11-93-18; e-mail: piter-sib@risp.ru

Ростов-на-Дону ул. Калитвинская, д. 17в; тел. (8632) 95-36-31, (8632) 95-36-32; e-mail: jupiter@rost.ru

Самара ул. Новосадовая, д. 4; тел. (8462)37-06-07; e-mail: piter-volga@sama.ru

УКРАИНА

Харьков ул. Суздальские ряды, д. 12, офис 10–11, т. (057) 712-27-05, 712-40-88; e-mail: piter@tender.kharkov.ua

Киев пр. Красных Казаков, д. 6, корп. 1; тел./факс (044) 490-35-68, 490-35-69; e-mail: office@piter-press.kiev.ua

БЕЛАРУСЬ

Минск ул. Бобруйская д., 21, офис 3; тел./факс (37517) 226-19-53; e-mail: piter@mail.by

---

Ищем зарубежных партнеров или посредников, имеющих выход на зарубежный рынок.
Телефон для связи: **(812) 103-73-73**.
E-mail: grigorjan@piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов.
Обращайтесь по телефонам: **Санкт-Петербург – (812) 327-13-11**,
Москва – (095) 777-54-67.

Заказ книг для вузов и библиотек: (812) 103-73-73.
Специальное предложение – e-mail: kozin@piter.com



УВАЖАЕМЫЕ ГОСПОДА!
КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
ВЫ МОЖЕТЕ ПРИОБРЕСТИ
ОПТОМ И В РОЗНИЦУ
У НАШИХ РЕГИОНАЛЬНЫХ ПАРТНЕРОВ.

Башкортостан

Уфа, «Азия», ул. Зенцова, д. 70 (оптовая продажа),
маг. «Оазис», ул. Чернышевского, д. 88,
тел./факс (3472) 50-39-00.
E-mail: asiaufa@ufanet.ru

Дальний Восток

Владивосток, «Приморский торговый дом книги»,
тел./факс (4232) 23-82-12.
E-mail: bookbase@mail.primorye.ru

Хабаровск, «Мирс»,
тел. (4212) 30-54-47, факс 22-73-30.
E-mail: sale_book@bookmirs.khv.ru

Хабаровск, «Книжный мир»,
тел. (4212) 32-85-51, факс 32-82-50.
E-mail: postmaster@worldbooks.kht.ru

Европейские регионы России

Архангельск, «Дом книги»,
тел. (8182) 65-41-34, факс 65-41-34.
E-mail: book@atnet.ru

Калининград, «Вестер»,
тел./факс (0112) 21-56-28, 21-62-07.
E-mail: nshibkova@vester.ru
http://www.vester.ru

Северный Кавказ

Ессентуки, «Россы», ул. Октябрьская, 424,
тел./факс (87934) 6-93-09.
E-mail: rossy@kmw.ru

Сибирь

Иркутск, «ПродаЛитЪ»,
тел. (3952) 59-13-70, факс 51-30-70.
E-mail: prodalit@irk.ru
http://www.prodalit.irk.ru

Иркутск, «Антей-книга»,
тел./факс (3952) 33-42-47.
E-mail: antey@irk.ru

Красноярск, «Книжный мир»,
тел./факс (3912) 27-39-71.
E-mail: book-world@public.krasnet.ru

Нижневартовск, «Дом книги»,
тел. (3466) 23-27-14, факс 23-59-50.
E-mail: book@nvartovsk.wsnet.ru

Новосибирск, «Топ-книга»,
тел. (3832) 36-10-26, факс 36-10-27.
E-mail: office@top-kniga.ru
http://www.top-kniga.ru

Тюмень, «Друг»,
тел./факс (3452) 21-34-82.
E-mail: drug@tyumen.ru

Тюмень, «Фолиант»,
тел. (3452) 27-36-06, факс 27-36-11.
E-mail: foliant@tyumen.ru

Челябинск, ТД «Эврика», ул. Барбюса, д. 61,
тел./факс (3512) 52-49-23.
E-mail:evrika@chel.surnet.ru

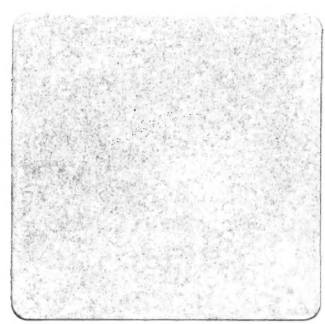
Татарстан

Казань, «Таис»,
тел. (8432) 72-34-55, факс 72-27-82.
E-mail: tais@bancorp.ru

Урал

Екатеринбург, магазин № 14,
ул. Челюскинцев, д. 23,
тел./факс (3432) 53-24-90.
E-mail: gvardia@mail.ur.ru

Екатеринбург, «Валео-книга»,
ул. Ключевская, д. 5,
тел./факс (3432) 42-56-00.
E-mail: valeo@etel.ru



Э. ТАНЕНБАУМ

АРХИТЕКТУРА КОМПЬЮТЕРА

4-Е ИЗДАНИЕ

КНИГИ, КОТОРЫЕ НЕ СТАРЕЮТ!

КЛАССИКА COMPUTER SCIENCE

Компьютер не знает иного языка, кроме машинного, состоящего из довольно примитивного набора команд, которые придумывают разработчики процессора. Именно этими командами приходилось пользоваться первым программистам, чтобы заставить компьютер что-то выполнить. Этот машинный язык сохранился и в наши дни, но теперь между ядром компьютера и приложениями возникли многочисленные посредники в виде микропрограмм, операционных систем и языков программирования высокого уровня.

Независимо от конкретных типов процессора и операционной системы, современный компьютер можно рассматривать как абстрактную многоуровневую иерархическую систему, каждый уровень которой выполняет определенные типовые функции. В этой книге описываются самые базовые принципы организации компьютера, что позволяет читателю получить фундаментальное представление о его работе.

В четвертом издании структура книги в целом сохранилась, но содержание обновилось, отражая изменения в компьютерных технологиях. Например, все примеры программ, которые в предыдущих изданиях были написаны на языке Pascal, в четвертом издании переписаны на языке Java, популярном в последнее время. При описании аппаратной части компьютера рассматриваются более современные устройства ввода-вывода. В книге затрагивается широкий круг вопросов: от мультипроцессоров до кластерных систем, поэтому материал, связанный с архитектурами параллельного действия, был полностью переделан и значительно расширен.

ISBN 5-318-00298-6



Посетите наш web-магазин: www.piter.com

ПИТЕР®
WWW.PITER.COM

PH PTR

CS

4-Е ИЗДАНИЕ

Э. ТАНЕНБАУМ

АРХИТЕКТУРА КОМПЬЮТЕРА

PH PTR